**MUTEX VS SEMAPHORE**

**Problems with Semaphores -**  
  
1) Accidental release  
This problem arises mainly due to a bug fix or cut-and-paste mistake. In this case, a semaphore isn’t correctly acquired but is then released.  
  
/\* Oops, forgot P(S) \*/  
...  
critical section code  
...  
V(S)  
  
Each time the buggy code is executed the count is increment and yet another task can enter the critical section.  
  
2) Deadlock  
  
2.1) Recursive Deadlock  
Recursive deadlock can occur if a task tries to lock a semaphore it has already locked.  
  
myFunction() {  
  
P(S)  
...  
critical section code starts  
...  
myFunction();  
...  
critical section code ends.  
...  
V(S)  
  
Here the S was still 0 when again recursive call was made.  
  
2.2) Deadlock through Task Death  
What if a task that is holding a semaphore dies or is terminated? If you can’t detect this condition then all tasks waiting (or may wait in the future) will never acquire the semaphore and deadlock.  
  
2.3) Priority Inversion  
Priority inversion is the case where a high priority task becomes blocked for an indefinite period by a low priority task. Lets say a thread A was waiting for completion of thread B to proceed. In the mean time, thread C with priority more than B came in and pre-empted thread B - thus thread A never completed.  
  
2.4) Semaphore as a Signal  
Assuming Task1 calls theP(S) it will block. When Task 2 later calls the V(S) then the unilateral synchronization takes place and both task are ready to run  
  
**MUTEX - a little bit more...**  
  
The mutex is similar to the principles of the binary semaphore with one significant difference: the principle of ownership. Ownership is the simple concept that when a task locks (acquires) a mutex - only it can unlock (release) it. If a task tries to unlock a mutex it hasn’t locked (thus doesn’t own) then an error condition is encountered and, most importantly, the mutex is not unlocked. It solves - all the problems with Semaphores listed above -  
  
1) Accidental Release  
Ownership stops accidental release of a mutex as a check is made on the release and an error is raised if current task is not owner.  
  
2.1) Recursive Deadlock  
Due to ownership, a mutex can support relocking of the same mutex by the owning task as long as it is released the same number of times.  
  
2.2) Priority Inversion  
With ownership this problem can be addressed using one of the following priority inheritance protocols. The idea is never let a high priority thread pre-empt you:  
  
a) The Basic Priority Inheritance Protocol enables a low-priority task to inherit a higher-priorities task’s priority if this higher-priority task becomes blocked waiting on a mutex currently owned by the low-priority task. The low priority task can now run and unlock the mutex – at this point it is returned back to its original priority.  
  
b) With the Priority Ceiling Protocol (PCP) method each mutex has a defined priority ceiling, set to that of the highest priority task which uses the mutex. Any task using a mutex executes at its own priority – until a second task attempts to acquire the mutex. At this point it has its priority raised to the ceiling value, preventing suspension and thus eliminating the “hold and wait” condition.  
  
2.3) Death Detection  
If a task terminates for any reason, the RTOS can detect if that task current owns a mutex and signal waiting tasks of this condition. In terms of what happens to the waiting tasks, there are various models, but two dominate:  
a) All tasks readied with error condition;  
b) Only one task readied; this task is responsible for ensuring integrity of critical region.  
  
When all tasks are readied, these tasks must then assume critical region is in an undefined state. In this model no task currently has ownership of the mutex. The mutex is in an undefined state (and cannot be locked) and must be reinitialized.  
  
When only one task is readied, ownership of the mutex is passed from the terminated task to the readied task. This task is now responsible for ensuring integrity of critical region, and can unlock the mutex as normal.  
  
**Some known APIs**  
  
It should be noted that many Real-Time Operating Systems (or more correctly Real-Time Kernels) have their own logics and APIs for mutexes and semaphores. Some common RTOS are -  
  
VxWorks Version 5.4  
POSIX Threads (pThreads) – IEEE Std 1003.1, 2004 Edition  
Microsoft Windows Win32 – Not .NET