embedded-os.de
a little world of RTOS and data-communication protocols

Special

Priority Inversion

Haftungsausschluß
Der Autor übernimmt keinerlei Haftung für durch diesen Code entstandene oder entstehende Schäden an Hard- und Software. Er versichert lediglich, daß er den Code vielfältigen Test´s auf unterschiedlicher Hardware unterzogen hat, um seinerseits keine Fehler bestehen zu wissen. Sollten dennoch Fehler auftauchen oder Vorschläge zur Verbesserung des Codes an den Autor weitergegeben werden, so ist dieser bestrebt, Fehler schnellstmöglich auszumerzen oder Vorschläge einzuarbeiten.

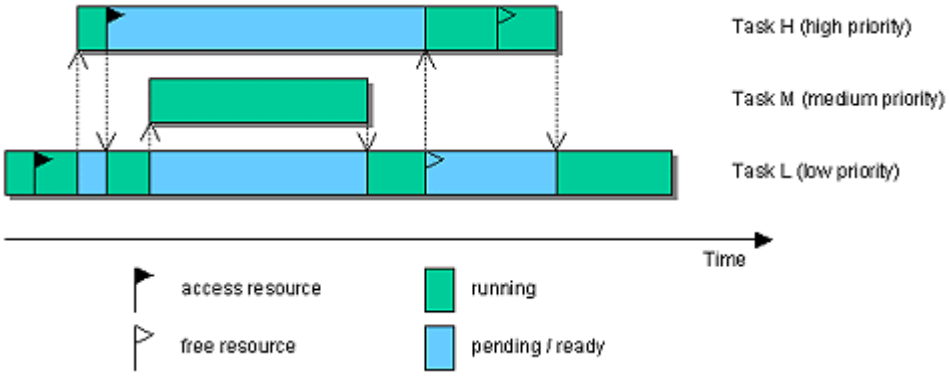
liability exclusion;
The author takes over no liability for through this code originated or emerging damages to hardware and software. He assures merely that he subjected the code of diverse tests on different hardware, about for his part no mistakes to know exists. Mistakes nevertheless should appear or suggestions are passed on at the author to the improvement of the code, so this is striving, mistakes fastest to wipe out or to incorporate suggestions.

Priority Inversion, das Problem und die Lösungsansätze

von Frank Götze (c)2006 - http://www.embedded-os.de

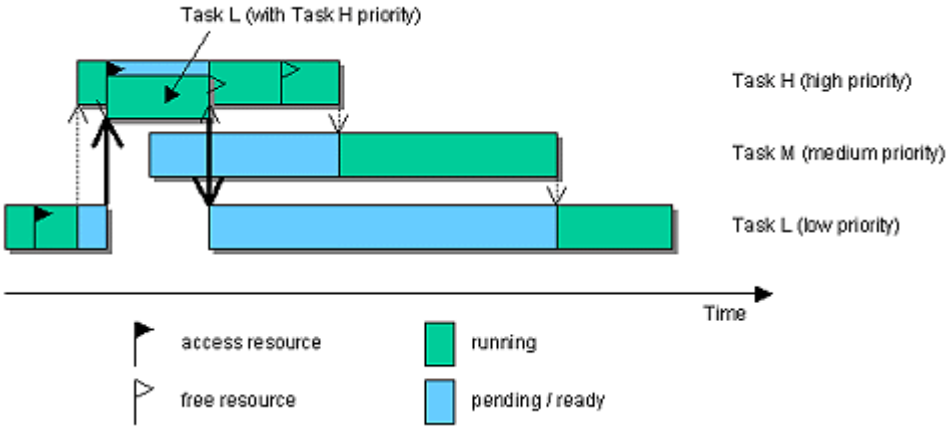
Priority Inversion **(Prioritätsumkehr) nennt man den Zeitraum, in dem ein niederpriorisierter Task vor einem höherpriorisierten Task ausgeführt wird. Wie kann so etwas zu Stande kommen ?**

Belegt ein niederpriorisierter Task eine Resource, z.B. mittels einer binären Semaphore, und will ein hochpriorisierter Task die selbe Resource während dieser Zeit belegen, so muß dieser hochpriorisierte Task warten, bis die Resource wieder frei ist.
-- so weit so gut --
Wird aber innerhalb dieses Zeitraumes, in dem der hochpriorisierte Task wartet, ein mittelpriorisierter Task aktiv (z.B. durch einen Interrupt o.ä.), so wird der niederpriorisierte Task, der die Resource besitzt, unterbrochen und der mittelpriorisierte Task ausgeführt, obwohl ein hochpriorisierter Task dringend auf die besetzte Resource wartet.



Dies läßt sich auch noch um weitere Tasks M1...Mx mittlerer Priorität erweitern, die bei einer höheren Priorität als Task M, diesen zu seiner Laufzeit und sich gegenseitig unterbrechen und auf Grund ihrer jeweiligen Prioritäten zur Ausführung kommen.
All dies muß nicht zwangsläufig zu einem schweren Problem führen, solange keine Zeitlimits (deadlines) überschritten werden, kann aber wie im Fall der Mars Mission Pathfinder im Juli 1997 zu schweren Bildübertragungsproblemen und, bei Verwendung eines Watchdog-Timers, zusätzlich zu System-Resets führen.
Für dieses Problem gibt es zwei Lösungsansätze: *Priority Inheritance* und *Priority Ceiling*.

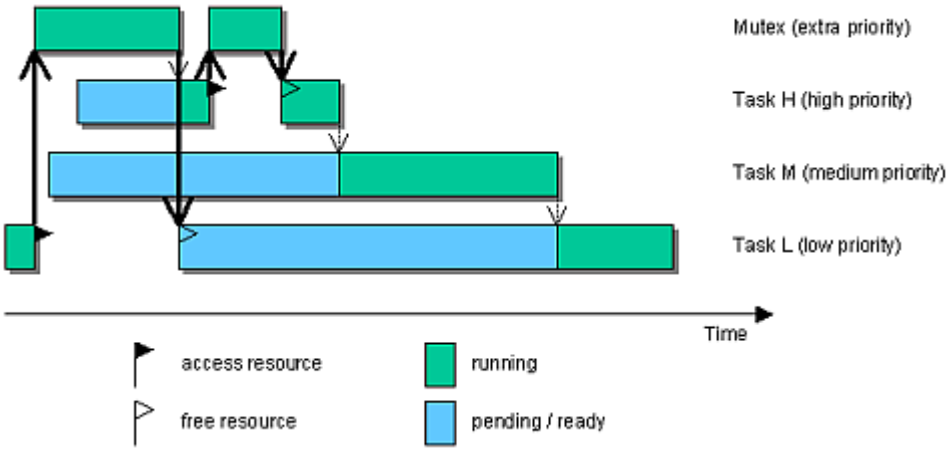
Priority Inheritance **(Prioritätserbschaft) nennt man die Lösung, bei der mittels einer MUTEX (Mutual Exclusion) der niederpriorisierte Task auf die selbe Priorität angehoben wird wie der hochpriorisierte Task, wenn dieser die Resource belegen will. Das setzt aber vorraus, daß der verwendete Kernel mehrere Tasks mit der selben Priorität verwalten kann und solche Prioritätsänderungen auch dynamisch vor und zurück durchführen kann. Desweiteren kann dies zu einer höheren Anzahl von Task-Switches führen. Außerdem kann es kompliziert werden, wenn ein Task die Resource A besitzt und nun noch eine zweite Resource B belegen will... siehe gegenseitige Blockade wenn diese Resource B durch einen auf Resource A wartenden Task bereits besetzt ist.**



Kommen zusätzlich Tasks H1..Hx noch ins Spiel, so wird der Task, der die Resource besitzt, immer auf die jeweils höchste Priorität der wartenden Tasks angehoben. Steigt der höchstpriorisierte dieser Tasks mit einem TimeOut aus der Warteschlange aus, so sollte der Task L nun die Priorität des verbleibenden höchstpriorisierten wartenden Tasks erhalten, um höherpriorisierte Tasks, die nicht auf diese Recource zugreifen wollen, zuzulassen.

Zusammenfassung *Priority Inheritance* :
+ dynamisch
+ unbekannte Tasks und ihre Prioritäten möglich
+ mittelpriorisierte Tasks können aktiv werden, wenn kein hochpriorisierter Task auf die Resource wartet
- relativ aufwendige Verwaltung durch das OS
- Gefahr von Deadlocks gegeben (Resource A - Resource B)
- höhere Anzahl an Prioritätswechseln
- erhöhte Anzahl an Task-Switches möglich

Priority Ceiling **(Prioritätsdecke) nennt man die Lösung, bei der mittels einer MUTEX (Mutual Exlusion) mit eigener Priorität, die höher als alle Task-Prioritäten der auf die Recource zugreifenden Tasks sein muß, garantiert wird, daß bei Belegung der Resource kein mittelpriorisierter Task aktiv werden kann. Jedoch kann bei sofortiger Prioritätsanhebung auch kein mittelpriorisierter Task aktiv werden, wenn kein hochpriorisierter Task auf die Resource wartet. Desweiteren müssen bei dieser Lösung alle Tasks und ihre Prioritäten, die auf die Resource zugreifen können/wollen zu compile-Zeit bekannt sein.**



Zusammenfassung *Priority Ceiling* :

+ sehr sicher bei Bekanntsein aller Tasks die die Resource belegen können/wollen
+ einfache Verwaltung
+ geringe Anzahl an Prioritätswechseln
+ keine erhöhte Anzahl an Task-Switches
+ Gefahr von Deadlocks kontrollierbar durch die MUTEX Prioritäten (Resource A - Resource B)
- unbekannte Tasks und ihre Prioritäten unmöglich
- bei sofortiger Prioritätsanhebung zum Zeitpunkt der Belegung der Resource kann kein mittelpriorisierter Task aktiv werden, auch wenn kein hochpriorisierter Task auf die Resource wartet

Lösung im pC/OS :

Im pC/OS-Kernel wird ausschließlich das Verfahren *Priority Ceiling* verwendet, da dieser Kernel mehrere Tasks mit gleicher Priorität nicht verwalten kann. Somit müssen alle Tasks, die die jeweilige Resource belegen können/wollen, zu compile-Zeit bekannt sein. Die verhältnismäßig einfache Verwaltung kommt dem Kernel dafür sehr entgegen. Eine Adaption der Zeitpunkte der Prioritätsanhebung und Wiederabsenkung des Tasks, der die Resource belegt hat, wurde nach einigen Überlegungen verworfen. Dabei sollte die Priorität des Tasks erst angehoben werden, wenn ein höher priorisierter Task die Resource belegen will und die Wiederabsenkung der Priorität des Tasks durchgeführt werden, wenn der letzte höherpriorisierte Task sein Warten auf diese Resource abbricht (TimeOut) oder der Task selbst die Resource wieder frei gibt. Dies hätte jedoch der Gefahr von Deadlocks (Resource A - Resource B) wieder alle Türen geöffnet.

Literaturverweis:

(1) Introduction to Priority Inversion (by Michael Barr - Embedded Systems Programming)
(2) Priority inversion scenario with a binary semaphore (by David Kalinsky - Enea OSE Systems)
(3) RTOSes, ´mutexes´ fight priority inversion (by David Kalinsky - Enea OSE Systems)
(4) google.de - Suche nach den Keywords

Comments

Comments