RTOS Einführung

Version: 0.0.1
Datum: 20.07.2013
Autor: Werner Dichler

Inhalt

RTOS

Definition

RTOS = Real Time Operating System

Der Begriff Echtzeit beschreibt in der Datenverarbeitung die Eigenschaft, dass bei der Eingabe, Verarbeitung und Ausgabe die zeitlichen Anforderungen, welche durch das System vorgegeben sind, erfüllt werden müssen.

Anforderungen

- Rechtzeitigkeit ... unabhängig von der Schnelligkeit
- Gleichzeitigkeit ... scheinbar gleichzeitiges Abarbeiten von Tasks
- Determiniertheit ... bei jeder Kombination von Eingangsgrößen ist die Reaktionszeit in festen zeitlichen Grenzen vorhersagbar
- Unterbrechbar durch Interrupts und Traps

$$U = \sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

m ... Anzahl der periodischen Ereignisse
P_i ... Auftritts-Periode des Ereignisses i
C_i ... Abarbeitungs-Zeit des Ereignisses i

Formel 1 – Echtzeitfähigkeit

Ein System ist Echtzeitfähig wenn obige Forderung erfüllt wird. Wenn z.B. drei Ereignisse mit den Eigenschaften unterhalb verarbeitet werden müssen, ergibt sich ein Grad der Auslastung U von 0.85. Das bedeutet, dass das System noch Echtzeitfähig ist.

P_1 = 100ms, C_1 = 50ms	U_1 = 0.5
P_2 = 200ms, C_2 = 30ms	U_2 = 0.15
P_3 = 500ms, C_3 = 100ms	U_3 = 0.2

Aufgaben

Ein echtzeitfähiges Betriebssystem sollte die oben genannten Anforderungen erfüllen. Für die Erfüllung ergeben sich einige Aufgaben. Meistens wird mit einem Betriebssystem des Weiteren versucht, das System mit einer Architektur zu versehen. Somit entstehen weitere Abstrahierungsstufen, welche bei großen Systemen den Überblick fördern.

- Taskverwaltung ... ausführen, anlegen, löschen
- Speicherverwaltung ... anfordern, freigeben
- Synchronisationsmöglichkeiten ... Unterbrechungen unterbinden
- Kommunikationsmöglichkeiten ... Kommunikation zwischen Tasks
- Abstrahierung der Hardware ... Hardware Abstraction Layer (HAL), Treiber

Eigenschaften

Bei der Auswahl eines Betriebssystems bzw. bei der Implementierung eines eigenen Betriebssystems ist der Overhead, der durch demselben verursacht wird, ein wichtiges Kriterium.
Große und mächtige Betriebssysteme haben oft den Nachteil, dass sie selbst viele Ressourcen in Anspruch nehmen. Dem gegenüber sind kleinere Betriebssysteme Leichtgewichte. Diese bieten dafür keine umfangreiche Funktionalität, welche die Entwicklungsarbeit erleichtern würde.

- Programmspeicher-Bedarf
- RAM-Bedarf
- Dauer eines Task-Wechsels

Einteilung der Betriebssysteme

Synchrone Programmierung

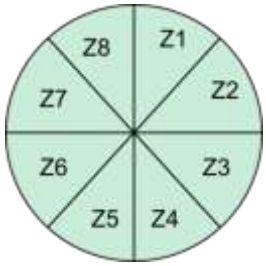

Bild 1 – Synchrone Programmierung

Die Verarbeitung wird in Zeitscheiben aufgeteilt. Jede Zeitscheibe hat eine definierte maximale Dauer und einen bestimmten Start-zeitpunkt. Die Ausführung der Teilprogramme darf die Dauer einer Zeitscheibe nicht über-schreiten.

Kooperatives Multitasking

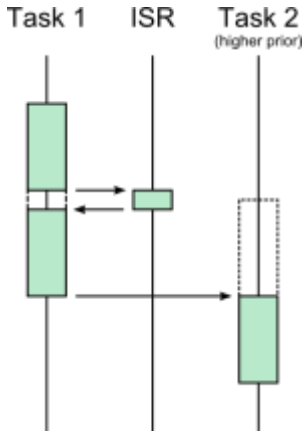

Bild 2 – kooperatives Multitasking

Die Verarbeitung wird in Tasks aufgeteilt. Dabei muss ein Task-Wechsel vom aktuell laufenden Task zugelassen werden.

Preemptives Multitasking

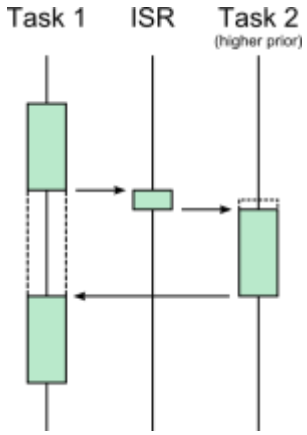

Bild 3 – preemtives Multitasking

Die Verarbeitung wird in Tasks aufgeteilt. Jeder Task kann jeder Zeit durch höher-priore Tasks unterbrochen werden.

Kernel Design

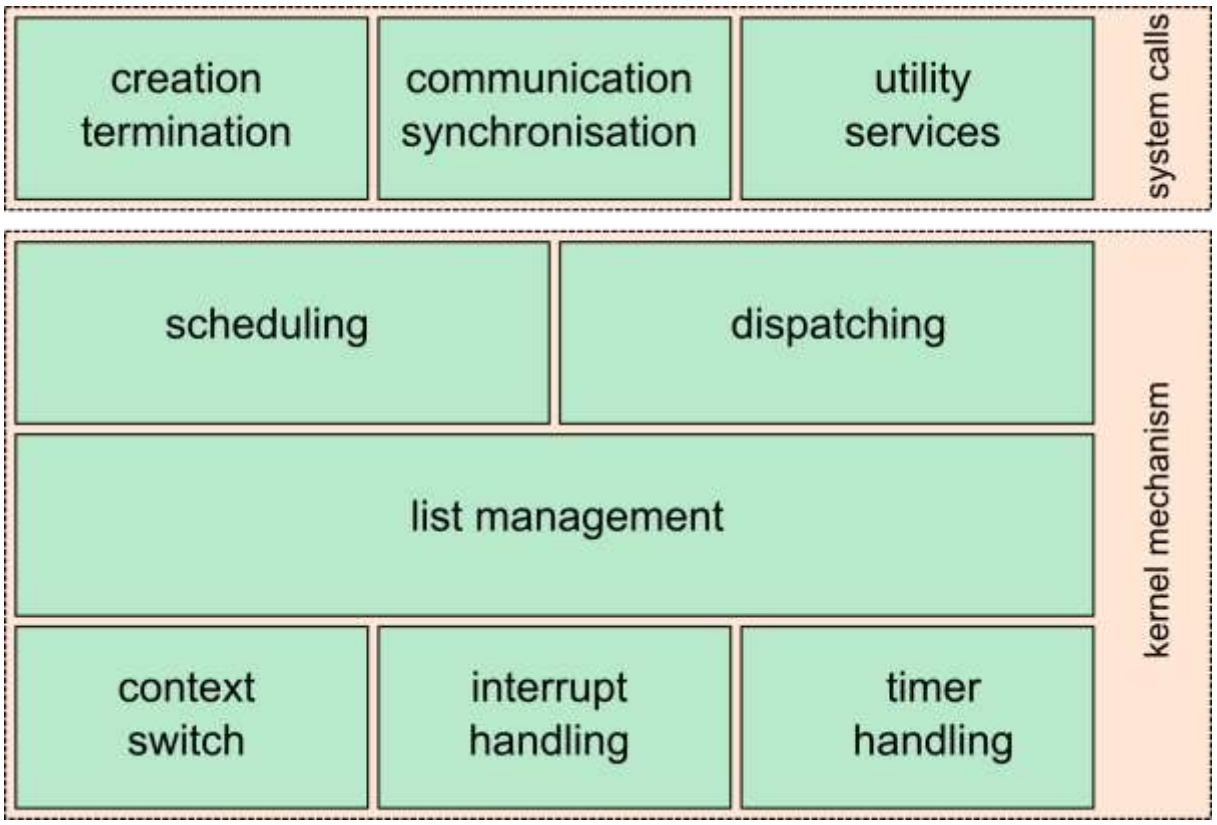

Bild 4 – Kernel Design

Dispatcher

Der Dispatcher übergibt den nächsten Task, der als nächster für die Ausführung ausgewählt wurde, die Kontrolle über den Prozessor. Vor dem Task Wechsel muss der zuvor laufende Tasks gesichert werden (Stack, Register und Position im Programm).

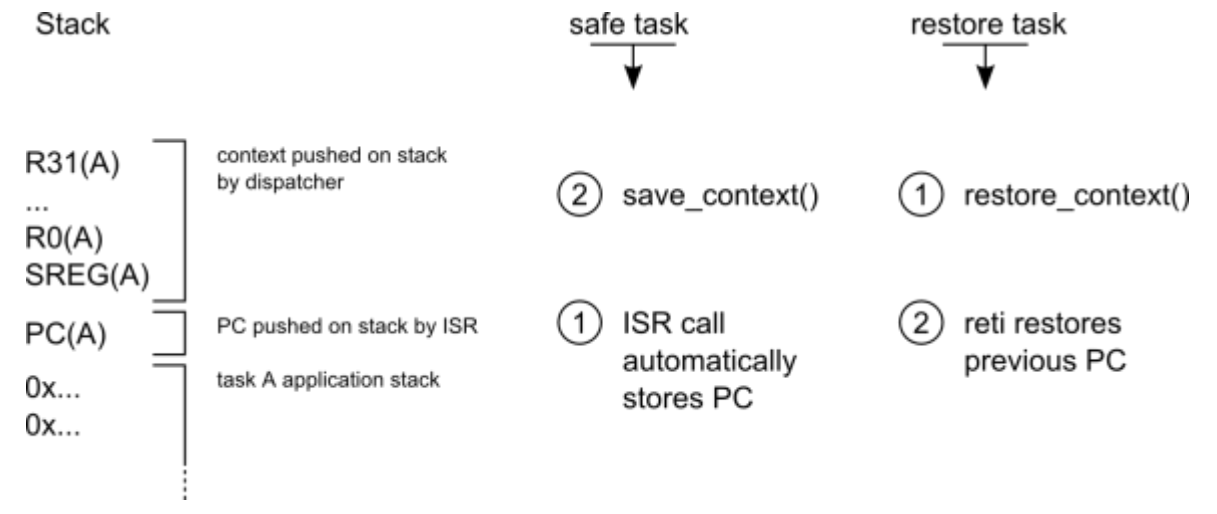

Bild 5 – Bsp. für Task Kontext speichern/wiederherstellen

Task Zustände

Ein aktiver Task kann unterschiedliche Zustände annehmen. Mit der Information des Zustandes kann das Betriebssystem die Ausführung besser organisieren. Somit wird ein Task, der auf ein Ereignis wartet, nicht aufgerufen, solange das Ereignis nicht eingetreten ist.

- Laufend (Running) ... wird zur Zeit ausgeführt
- Bereit (Ready) ... zur Ausführung bereit
- Wartend (Waiting) ... wartet auf ein Ereignis, Ressource, ...
- Ruhend (Idle) ... Periode abgeschlossen, wartet auf nächste Periode

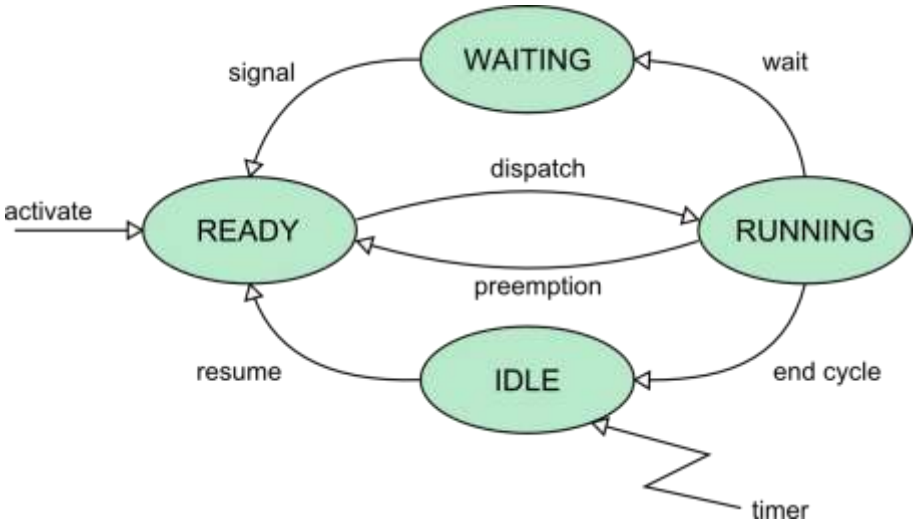

Bild 6 – Zustandsübergänge

Scheduling Mechanismen

Der Scheduler ist für das optimale Zuteilen des Prozessors zuständig. Die zur Ausführung bereiten Tasks werden nach unterschiedlichen Kriterien priorisiert (zugewiesene Priorität in Verbindung mit weiteren Eigenschaften). Grundsätzlich sollten die Tasks mit hoher Wichtigkeit schnell abgearbeitet werden. Des Weiteren sollten keine Tasks zu lange nicht abgearbeitet werden, also nicht verhungern.

- Earliest Deadline First ... Task mit frühesten Deadline hat höchste Priorität (Deadline wird im Design festgelegt)
- Rate Monotonic Scheduling ... kürzeste Periode hat höchste Priorität (Periode und Laufzeit wird im Design festgelegt)
- Deadline Monotonic Scheduling ... Priorität aufgrund relativer Deadlines
- Round Robin Scheduling ... abwechselnde Abarbeitung

Ein statisches Scheduling ist bei einer Überlast stets vorhersagbar. Das dynamische Scheduling ermöglicht höhere Prozessor-Auslastungen, produziert aber ein unvorhersehbares Verhalten.

Synchronisation

Synchronisations-Mechanismen ermöglichen das Verriegeln eines Programmabschnittes, der nicht unterbrochen werden darf. Es gibt unterschiedliche Gründe, wodurch eine Verriegelung notwendig wird. Grundsätzlich wird sie benötigt, wenn zwei unterschiedliche Task auf die selbe Ressource zugreifen und eine Race Condition entstehen könnte.

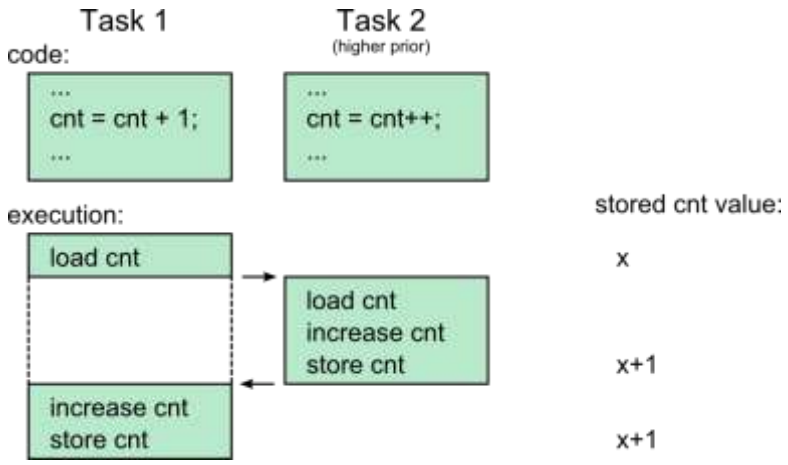

Bild 7 – Race Condition Beispiel

An dem Beispiel oberhalb kann man erkennen, dass eine Race Condition auf Ebene der Hochsprache nicht einfach zu identifizieren ist. Bei dem Beispiel wird die Zähler-Variable statt zweimal nur einmal erhöht, da der Task 1 den veralteten Wert erhöht und zurück schreibt.
Das Debuggen von einer Race Condition ist nur schwierig möglich. Das Auftreten einer Race Condition hängt stark von der Ausführungsfolge der Tasks ab. Da das Debuggen selbst das Laufzeitverhalten verändert, tritt die Race Condition nur mehr mit einer sehr geringen Wahrscheinlichkeit auf.

Semaphore / Mutex

Mit einem Semaphore kann eine Ressource in Anspruch genommen werden. Der Semaphore ist so Konfiguriert, dass die Ressource je nach Anzahl der gleichzeitigen Verfügbarkeit verwendet werden kann. Stellt die Ressource z.B. ein Puffer mit 5 Elementen dar, so kann die Ressource 5 mal in Anspruch genommen werden. Wurde die Ressource zur Gänze in Anspruch genommen, so wird der nächste Task informiert bzw. gleich ruhend gestellt.
Ein Mutex (Mutual Exclusion) ist ein Semaphore mit einem Zähler gleich 1. Also gleichzusetzen mit einem binären Semaphore.

- use() ... Beanspruchen einer Ressource;
  wenn nicht frei, wird Task ruhend gestellt
- unuse() ... Freigabe der Ressource
- request() ... Abfrage ob Ressource frei;
  wenn frei, wird die Ressource in Anspruch genommen

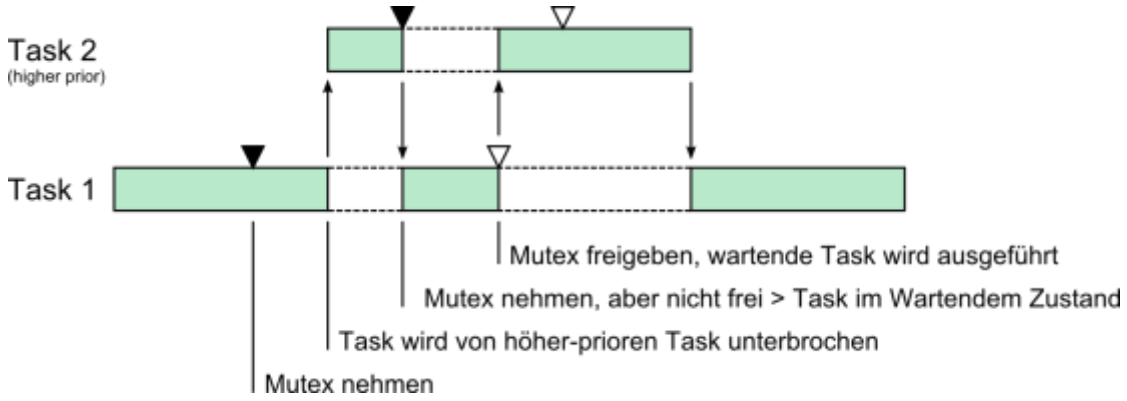

Bild 8 – Mutex

Critical Section

Bei der Critical Section wird üblicherweise der Scheduler deaktiviert. Somit kann der aktuell laufende Task von anderen höher-prioren Tasks nicht unterbrochen werden. Hardware Interrupts werden dennoch ausgeführt. Da innerhalb einer Critical Section der Scheduler zur Gänze deaktiviert ist, sollte der Einsatz auf möglichst kurze und zeitkritische Abschnitte beschränkt werden.
Eine Critical Section ist gegenüber einem Semaphore oder Mutex schneller und verursacht somit selbst weniger Overhead.

- enter_CS() ... eine Critical Section betreten
- leaf_CS() ... eine Critical Section verlassen

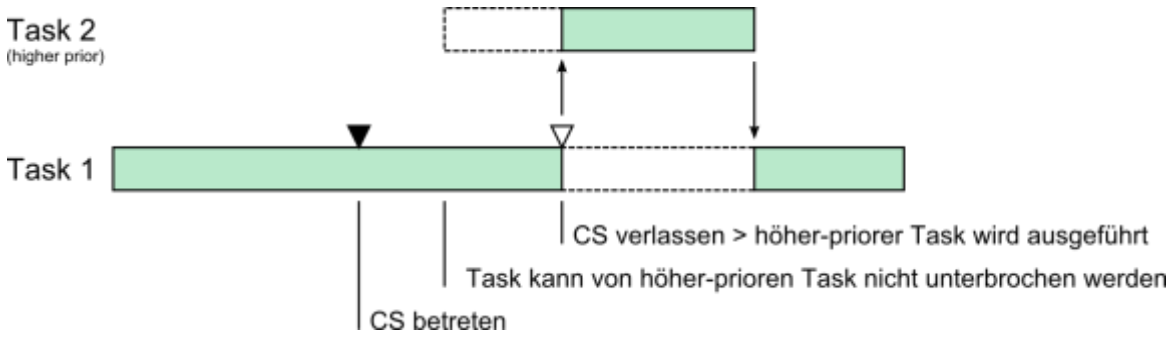

Bild 9 – Critical Section

Deadlock

Werden in einem Programmabschnitt mehrere Ressourcen gleichzeitig verwendet, so kann es zu einem zusätzlichen Problem führen - einem Deadlock. Bei einem Deadlock haben sich zwei Task, die die gleichen Ressourcen verwenden möchten, in einem Zustand gebracht, wo beide Tasks blockiert und nicht mehr lauffähig sind.

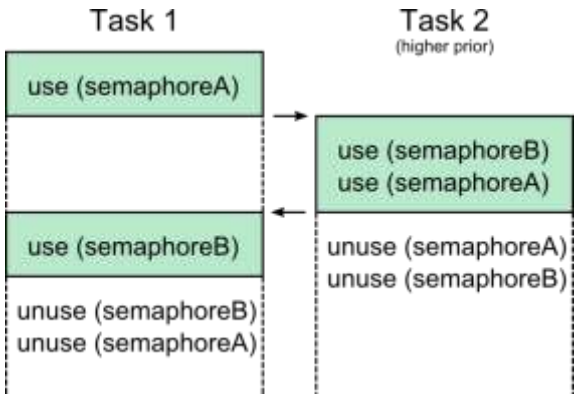

Bild 10 – Deadlock Beispiel

Deadlock Bedingungen:

- Eine Ressource kann genau einem Task zugeordnet sein bzw. noch verfügbar sein.
- Tasks können mehrere Ressourcen anfordern.
- Die Ressourcen können nur vom eigenen Task freigegeben werden.
- Es müssen mehrere Tasks auf eine Ressource des anderen Tasks warten.

Wird eine der Bedingungen für Deadlocks nicht erfüllt, so können sie einfach vermieden werden:

- Alle benutzten Ressourcen auf einmal anfordern.
- Ressourcen durchnummerieren und immer in gleicher Reihenfolge anfordern.
- Beim Anfordern immer nur eine Anfrage machen. Ist die Ressource nicht frei, werden sämtliche bereits angeforderten Ressourcen wieder freigegeben.

Prioritätsinversion

Blockiert ein nieder-priorer Task einen anderen höher-prioren Task durch die Verwendung einer Ressource, so kann die Dauer der Blockierung noch Verlängert werden, wenn ein etwas höher-priorer Task abgearbeitet wird. Somit kann es vorkommen, dass ein sehr wichtiger Task lange verzögert wird.

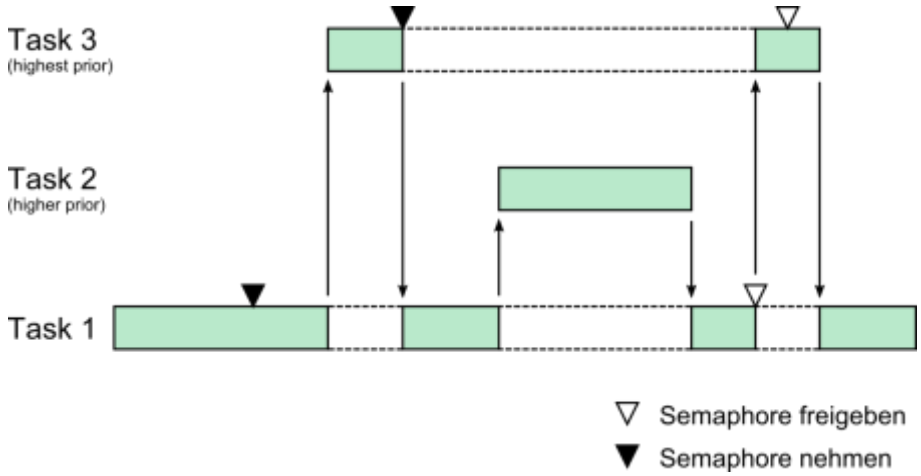

Bild 11 – Prioritätsinversion

Die Verzögerung kann Verhindert werden indem der niedrig-priore Task die Priorität des wartenden hoch-prioren Tasks übernimmt. Nachdem der zwischenzeitlich aufgestufte Task die belegte Ressource freigibt, verringert sich wieder dessen Priorität und er kann wieder von etwas höher-prioren Tasks unterbrochen werden. Diesen Vorgang nennt man Prioritätsvererbung.

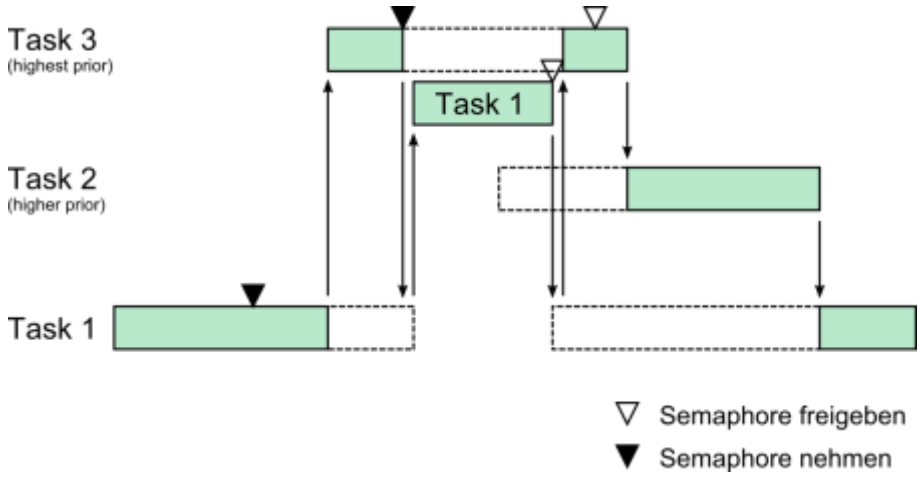

Bild 12 – Prioritätsvererbung

Kommunikation

Betriebssysteme bieten unterschiedliche Möglichkeiten an, um Informationen zwischen den Tasks auszutauschen. Die angebotenen Mechanismen sind oft bereits geschützt vor Race Conditions. Somit kann der zusätzliche Implementierungsaufwand minimieren werden, der andernfalls immer durchzuführen wäre.

Shared Memory

Ein Shared Memory ist ein bestimmter Speicherbereich, der von mehreren Tasks verwendet werden kann. Bei der Verwendung dieses Speichers ist der Zugriff nicht geregelt. Somit muss man selbst dafür sorgen, dass keine Race Condition entsteht. Ein Shared Memory kann als Block oder als Ring-Puffer ausgeführt werden.

Mailbox

Eine Mailbox wird vom Betriebssystem verwaltet und ist als FIFO ausgeführt. Die Größe und Anzahl der Mails ist begrenzt.

- CreateMB() ... Mailbox anlegen (Anzahl und Größe der Mails festlegen)
- PutMail() ... Mail eintragen
- GetMail() ... Mail abrufen

Message Queue

Eine Message Queue entspricht einer Mailbox, wobei ein Pointer übergeben wird. Somit können Nachrichten mit unterschiedlicher Größe ausgetauscht werden. Der Empfänger muss im Anschluss der Verwendung der Daten die Message löschen.

- CreateMsgQueue() ... Message Queue anlegen (Anzahl der Nachrichten festlegen)
- SendMsg() ... Nachricht versenden
- ReceiveMsg() ... Nachricht empfangen

Event

Mit Hilfe eines Events kann einem Task mitgeteilt werden, dass ein Ereignis aufgetreten ist. Ein Task der auf ein Ereignis wartet benötigt keine Prozessor-Ressourcen.

- SignalEvent() ... Signalisiert ein Event an einen ausgewählten Task
- WaitEvent() ... Warten auf einen/mehreren Event/s
- WaitEventTimed() ... Warten auf einen/mehreren Event/s, mit Timeout