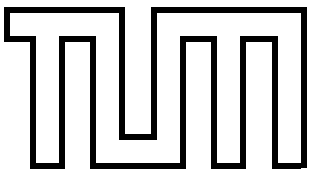

Technische Universität
München

Fakultät für Informatik
Forschungs- und Lehreinheit Informatik VI

Prozessrechner-Praktikum
Echtzeitsysteme

POSIX
(Threads, Semaphore, Nachrichtenwarteschlangen, Uhren)

Christian Buckl
buckl@in.tum.de

Matthias Regensburger
regensbu@in.tum.de

Dr. Gerhard Schrott
schrott@in.tum.de

Sommersemester 2008

1 POSIX

Im Rahmen dieses Praktikums sollen nach Möglichkeit nur POSIX-Funktionen verwendet werden. POSIX (**P**ortable **O**perating **S**ystem **I**nterface for Uni**X**) ist eine Sammlung von Betriebssystemfunktionen. Ziel von POSIX ist es eine allgemeine Schnittstelle zwischen Anwendungsapplikation und Betriebssystem bereitzustellen und so eine Portierung zwischen POSIX-konformen Betriebssystemen zu erleichtern.
Leider ist diese einfache Portierung nur unter Einschränkungen möglich. So unterstützen viele so genannte POSIX konforme Betriebssysteme nur eine eingeschränkte Auswahl von POSIX- Funktionen und es ergeben sich häufig leichte Abweichungen bei Signaturen und Namen der Funktionen. Trotzdem setzt sich der POSIX-Standard immer weiter durch und aus diesem Grund orientiert sich das Praktikum auch an diesem Standard. Ein weiterer Vorteil ist die Fülle an Informationen zu POSIX im Internet.

Ein guter Einstieg in POSIX sind die Internetseiten der OpenGroup:
http://www.opengroup.org/onlinepubs/007908799/index.html

Allgemein gibt es einige Konventionen zu den POSIX-Funktionen und Datentypen, die sehr charakteristisch sind:

- Datentypen werden in der Regel durch die Endung _t im Namen gekennzeichnet, z.B. pthread_t, sem_t
- Der typische Rückgabewert einer POSIX-Funktion ist ein Fehlercode. Ist der Rückgabewert 0, so konnte die Funktion ohne Fehler ausgeführt werden, ansonsten signalisiert der Wert die Art des Fehlers. Die Fehlercodes sind dabei in der Headerdatei errno.h spezifiziert. Eine Beschreibung finden sie in den Manuals zu VxWorks unter VxWorks Errno Code List. Die eigentlichen Rückgabewerte der Funktion werden über die Parameter zurückgegeben.

2 Threads

Echtzeitsysteme müssen typischerweise verschiedene Aufgaben gleichzeitig erledigen, die zudem noch unterschiedliche Anforderungen an die zeitliche Ausführung besitzen und von unterschiedlicher Wichtigkeit/Priorität sind. Aus diesem Grunde ist es sinnvoll, die Programme ebenfalls in unterschiedliche Abarbeitungsstränge zu unterteilen und die Programme mit Konzepten des Multitaskings bzw. Multithreadings zu implementieren.
Viele Betriebssysteme erlauben dem Entwickler die pseudoparallele Ausführung von Prozessen. Die Prozesse arbeiten jeweils eigenständig Programme ab und besitzen einen eigenen Speicherraum. Die Ausführung wird deshalb pseudoparallel genannt, da immer nur ein Prozess gleichzeitig auf der CPU ausgeführt werden kann. Echte Parallelität kann deshalb erst auf Multiprozessorensystemen erreicht werden.

Threads sind im Gegensatz zu Prozessen so genannte leichtgewichtige Prozesse, d.h. sie haben keinen eigenen Adressraum. Aus diesem Grund sind sie immer einer Hierarchie unterlegen. Alle Threads besitzen einen Vaterprozess, in dessen Speicherraum sie ausgeführt werden, bei Beendigung dieses Vaterprozesses wird auch die Ausführung der Threads beendet. Vorteil von Threads ist daher das effizientere Anlegen und der effektivere Kontextwechsel zwischen Threads.
Die Ausführung von Threads erfolgt Betriebssystem abhängig: entweder jeder einzelne Thread wird als eigenständig betrachtet und bekommt seine eigene Rechenzeit oder der Scheduler berücksichtigt die Hierarchien der Threads und verteilt die CPU gleichmässig unter den "Thread-Familien".

2.1 Threads in VxWorks

Threads werden in VxWorks nicht direkt unterstützt. Vielmehr gibt es in VxWorks genau genommen nur Tasks (also Prozesse). Dennoch wird die POSIX-Funktionalität für Threads unterstützt und auch wir benutzen diese Funktionalität.
Für den Benutzer ergeben sich nur marginale Unterschiede: so führt die Beendigung eines Vaterprozesses nicht zur automatischen Beendigung der vom ihm erzeugten Prozesse. Aufgrund der Tatsache, dass VxWorks keine Memory Management Unit besitzt, können Prozesse ohne Einschränkungen auch auf andere Speicherbereiche zugreifen. Dies hat einerseits Vorteile, da eine gemeinsame Kommunikation über den Speicher stark erleichtert wird, andererseits führt es zu einer nicht zu unterschätzenden Fehlerquelle.

2.2 Threads in POSIX

Speziell zu Threads findet sich eine Beschreibung aller relevanter Informationen unter:
http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

Eine Erläuterung der Unterschiede zwischen VxWorks und POSIX, sowie eine Beschreibung der von VxWorks unterstützten POSIX-Funktionen finden sie in den Manuals: VxWorks Programmer's Guide -> 3 POSIX Standard Interfaces

Die Bibliothek zum Umgang mit Threads ist sehr umfangreich und aus diesem Grunde ist eine Erläuterung aller Funktionen nicht sinnvoll. Deshalb bieten wir in diesem Abschnitt nur einen groben Überblick über die Funktionalität und verweisen auf die oben angegebenen Links für Detailwissen.

Die wichtigste Funktion zur Erzeugung von Threads ist *pthread_create( )* zur Erzeugung eines Threads.
Die Eigenschaften von Threads in Bezug auf die Verwaltung und das Scheduling können über Attribute spezifiziert werden, die über den Datentyp *pthread_attr_t* verwaltet werden.

Die entsprechenden Funktionen starten mit *pthread_attr_*.
Eine weitere wichtige Funktion ist die Funktion *pthread_join()*. Mit dieser Funktion kann auf die Beendigung eines anderen Threads gewartet werden.

2.3 Scheduling von Threads

Zum Scheduling, also der Strategie zur Zuteilung der Rechenzeit, stehen in VxWorks zwei verschiedene Verfahren zur Verfügung: Das prioritätenbasiertes FIFO- (first in first out) und das prioritätenbasierte RoundRobin-Verfahren. Bei beiden Verfahren werden den Prozessen durch den Entwickler Prioritäten zugeordnet. Grundsätzlich gilt, dass Prozesse mit höherer Priorität immer gegenüber Prozessen niederer Priorität bevorzugt werden, solange sie laufbereit sind.
Ein Prozess kann durch die Blockade eines Betriebsmittels (z.B. ein Semaphor, Warten auf ein Signal) blockiert sein. In diesem Fall wird der Prozess während der Blockade durch den Scheduler nicht berücksichtigt.
Die beiden oben genannten Verfahren unterscheiden sich in ihrer Strategie nur beim Management von Prozessen gleicher Priorität. Beim FIFO-Verfahren wird dem ersten laufbereiten Prozess höchster Priorität der Prozessor zugeteilt bis die Abarbeitung des Prozesses beendet wird, der Prozess blockiert oder ein Prozess höherer Priorität laufbereit wird.
Beim RR-Verfahren wird die Rechenzeit auf dem Prozessor in gleichmäßige Zeitintervalle eingeteilt (z.B. 20 ms), die den Prozessen höchster Priorität abwechselnd zugeteilt werden.
Achtung: In diversen Betriebssystemen unterscheidet sich die Prioritätenverteilung. So ist beispielsweise in VxWorks ein Prioritätenbereich von 0-255 definiert, wobei 0 die höchste Priorität darstellt. In POSIX ist dagegen das Maximum des Prioritätenbereichs die maximale Priorität. Solange Sie nur POSIX-Funktionen benutzen, gilt die POSIX-konforme Prioritätenverteilung.

Die wichtigsten Funktionen zum Scheduling in POSIX sind:

- *pthread_setschedparam* (setzen der Scheduling-Parameter zur Laufzeit)
- *pthread_attr_setschedparam* (setzen der Parameter vor der Threadinitialisierung)
- *sched_get_priority_max* (liefert die maximale Priorität zurück)
- *sched_get_priority_min* (liefert die minimale Priorität zurück)

All diese Funktionen sind in den Bibliotheken *pthread.h* und *sched.h* beschrieben.
(http://www.opengroup.org/onlinepubs/007908799/xsh/sched.h.html)

3 Interprozesskommunikation

Typischerweise benötigen Prozesse/Threads Informationen von anderen Prozessen/ Threads, aus diesem Grund ist es nötig Mechanismen zur Kommunikation einzuführen. In diesem Abschnitt werden drei der wichtigsten Mechanismen erläutert: Kommunikation über Speicher bzw. Nachrichtenwarteschlangen und Signalisierung durch Semaphore.

3.1 Kommunikation über gemeinsamen Speicher

Das einfachste Mittel zur Realisierung der Kommunikation ist die Benutzung eines gemeinsamen Speicherbereichs. Da es in VxWorks keine Memory Management Unit gibt, ist die Realisierung sehr einfach. Über globale Variablen kann auf gemeinsamen Speicherbereich zugegriffen werden.
Allerdings hat diese Methode auch Nachteile:

- keine Signalisierung bei Eintreffen eines neuen Wertes
- keine Zugriffskontrolle und deshalb möglicherweise Korruption der Daten

Im Allgemeinen werden aus diesen Gründen globale Variablen nur eingeschränkt benutzt. Ein Beispiel zur Verwendung von Speicher zur Kommunikation sind Variablen, deren Wert sich nach der Initialisierung nicht mehr ändert (oder zumindest während eines Zeitraums nicht ändert, während dem mehrere Prozesse auf die Variablen zugreifen können). Beispiel ist die Verwendung von globalen Variablen zur Signalisierung von Prozess-IDs (siehe Aufgabe 0).

3.2 Kommunikation per Nachrichtenwarteschlangen

Nachrichtenwarteschlangen sind die logische Weiterentwicklung von globalen Variablen und bieten eine Lösung für die oben genannten Probleme. Warteschlangen sind Speicher, für die Lese- und Schreibrechte getrennt vergeben werden können. Die Kommunikation erfolgt typischerweise unidirektional. Gleichzeitig wird sichergestellt, dass das Schreiben und Lesen von Nachrichten atomar erfolgt und somit eine Korruption der Daten ausgeschlossen ist.
Warteschlangen bieten auch die Möglichkeit mehrere Werte abzuspeichern. Dadurch kann eine Entkopplung der beteiligten Prozesse erreicht werden. Bis zur maximalen Speichergrenze kann der sendende Prozess die Warteschlange mit Nachrichten (Werten) füllen, der empfangende Prozess kann sie zu einem beliebigen Zeitpunkt in der Reihenfolge des Auftretens lesen. Zudem besteht die Möglichkeit auf das Eintreffen einer Nachricht zu warten.

Nachrichtenwarteschlangen (message queue) in POSIX:
Die Funktionen zu Nachrichtenwarteschlangen sind in der Bibliothek *mqueue.h* definiert. Eine Beschreibung ist hier verfügbar:
http://www.opengroup.org/onlinepubs/007908799/xsh/mqueue.h.html

Eine Nachrichtenwarteschlange kann über den Namen geöffnet werden. Die Attribute werden ähnlich zu Threads in einer Struktur *mq_attr* verwaltet.
POSIX unterstützt zudem eine Priorisierung der Nachrichten über die Vergabe von Prioritäten.

3.3 Signalisierung durch Semaphore

Semaphore stellen einen der ältesten Mechanismen zur Realisierung der Interprozesskommunikation darf. Semaphore dienen dabei in erster Linie dem Schutz von Betriebsmitteln, die von mehreren Prozessen benutzt werden. Semaphore funktionieren dabei ähnlich wie Schlüssel: für das geschützte Betriebsmittel (z.B. eine Variable) gibt es nur eine begrenzte Anzahl an Schlüsseln. Nur Besitzer des Schlüssels können auf die Variable zugreifen.
Die Schlüssel können von den einzelnen Prozessen angefordert werden (down, get, wait) und sollten möglichst bald wieder zurückgegeben werden (up, push, wait). Ist momentan kein Schlüssel verfügbar, so kann der Entwickler spezifizieren, ob die Funktion *down()*, bis zur Verfügbarkeit eines Schlüssels blockiert oder ob sie erfolglos zurückkehrt. Wichtigste Eigenschaft von Semaphoren ist die Atomarität der Operationen *down()* und *up()*.
Typischerweise werden zum Schutz von Bereichen nur binäre Semaphore (Wert 0 oder 1) eingesetzt, aber es gibt durchaus auch Anwendungsbereiche für die zählende Semaphore (Wert ≥ 0) das geeignete Mittel sind.
Es ist auch möglich, dass ein Prozess ständig Schlüssel erzeugt (up), während ein anderer Prozess Schlüssel ausschließlich konsumiert (down). Dadurch kann eine Signalisierung von Ereignissen erfolgen. Mögliches Anwendungsgebiet: Prozess 1 erzeugt zyklisch ein Ergebnis, dass durch Prozess 2 verarbeitet werden soll. Dieses Szenario kann natürlich auch über Nachrichtenwarteschlangen realisiert werden, allerdings ist die Umsetzung mittels Semaphoren schneller, da Semaphore ein Mittel der wertlosen (keine zu übertragende Daten) Signalisierung sind.

Semaphore in POSIX:
Zur Benutzung von Semaphoren steht in POSIX die Bibliothek *semaphore.h* zur Verfügung. Die Funktionen werden unter folgendem Link beschrieben:
http://www.opengroup.org/onlinepubs/007908799/xsh/semaphore.h.html

Ähnlich wie bei Nachrichtenwarteschlangen können Semaphore über ihren Namen benutzt werden. Die wichtigsten Funktionen sind:

- *sem_open* (Öffnen eines Semaphors)

- *sem_wait* (Anfordern eines Semaphores, blockierend)
- *sem_trywait* (nicht blockierendes Anfordern)
- *sem_post* (Freigabe des Semaphors)

4 Uhren und Alarme

Um überhaupt Echtzeitprogramme implementieren zu können, sind Uhren und Alarme essentiell. Auch hier stehen diverse Funktionalitäten für VxWorks mit der POSIX-Bibliothek zur Verfügung. Die entsprechenden Funktionen der Bibliothek *time.h* ermöglichen das Lesen der Uhrzeit *clock_gettime()*, die Initialisierung und Verwaltung von Alarmen *timer_...()*, sowie das Verzögern der Ausführung einzelner Prozesse um eine bestimmte Zeitdauer (*nanosleep*). Die Beschreibung ist unter http://www.opengroup.org/onlinepubs/007908799/xsh/time.h.html zu finden.

Die Verwendung von Alarmen in POSIX ist etwas komplizierter. Der Alarm löst entweder einmalig oder periodisch ein Alarmsignal (SIGALRM) aus. Mit Hilfe der Funktion *sigwait* der Bibliothek *signal.h* kann auf ein solches Signal gewartet werden.
(http://www.opengroup.org/onlinepubs/007908799/xsh/signal.h.html)

Beispiele zum Umgang mit Alarmen und Signalen finden sie unter
http://mia.ece.uic.edu/~papers/WWW/books/posix4/DOCU_006.HTM und
http://mia.ece.uic.edu/~papers/WWW/books/posix4/DOCU_007.HTM

Anmerkung: Zur Verwaltung der Uhrzeit wird eine Uhr benötigt. VxWorks simuliert eine solche Uhr mit der Bezeichnung *CLOCK_REALTIME*. Keine Uhr geht unendlich genau, vielmehr erfolgt das Fortschalten der Uhr in diskreten Schritten. Diese Uhrenfrequenz kann durch VxWorks interne Funktionen festgelegt werden:

- clock_setres
- clock_getres

Eine Beschreibung dieser Funktionen finden sie in der Tornado-Dokumentation unter:
VxWorks API Reference -> OS Libraries -> clockLib