Multicore in Echtzeitsystemen (1)

Liyuan Zhang
Friedrich-Alexander-Universität Erlangen-Nürnberg
Liyuan.Zhang@e-technik.stud.uni-erlangen.de

ABSTRACT
Bisher sind die Prozessoranzahl des Systems immer schnell gewachsen, damit besser Performance kriegen. Auch in Echtzeitssystemen sollen künftig Mehrkernprozessoren zum Einsatz kommen. Mit neuen Hard – und Softwareunterstützungen können die Applikationen hoch Speedup erreichen, das ist besonder vorteilhaft für Echtzeitanwendungen, weil hier die Ausführenszeit der Aufgaben einen zentrale Rolle spielt. Gleichzeitig werden auch einige kritischen Probleme auftauchen, zum Beispiel Synchronisation und Zugriffskontrolle zwischen mehreren Prozessoren. Kann man klassische Ablaufplanungen und Zugriffskontrolle des Uniprozessorsystems weiter in den Multicoreumgebungen benutzen? Sind solche Methoden noch gültig in den neuen Situationen?

Stichwörte
Echtzeit, Scheduling, Multiprozessor

1. EINLEITUNG
Das Thema dieses Papiers geht um echtzeitfähiges Mehrkernscheduling. Dieser Artikel wird sich mit den Grundlagenproblemen bei Entwicklung der Echtzeitsystemen in den Multicoreumgebungen beschäftigen, besonder um die neuen Herausforderungen und Algorithmen der Ablaufplanungen zu erläutern.

Im Abschnitt 2 wird einen Überblick über die gebräuchlichen Schedulingsverfahren und Zugriffskontrolleverfahren in Uniprozessorumgebung aufgezeigt. Hierbei wird näher auf das Protokoll *Prioritätsobergrenzen* (engl. Priority Ceiling [1]) vorgestellt.

Im Abschnitt 3 wird zuerst die neuen Herausforderungen aufgezeigt, wann man sich Echtzeitanwedungen in Multiprozessorsysteme entwickelt. Dann wird zwei wichtige Ablaufplanungsalgorithmen erläutert, nämlich *Multiprocessor Priority Ceiling Protocol* (Abk MPCP [2]), anderer Algorithmus heißt *End-to-End Tasks Scheduling* [3]. Dritt Teil ist auch der Schwerpunkt dieses Papiers.

2. ECHTZEIT-SCHEDULING MIT UNIPROZESSOR
Erfolgreicher Ablaufplanungsentwurf ist ein sehr wichtiges Merkmal für effektive Echtzeitsysteme. Seit langer Zeit ist das Desgin eleganter Schedulingsverfahren ein spannendes Thema im Forschungsbereich. Natürlich können Hardware kaputt, dadurch wichtige Applikationen der Kunden verletzt werden, obwohl wir alle Aufgaben/Tasks sehr sorgfältig geplant haben, 100% Garantie gibt es nicht. Ein gültiges Scheduling bedeutet, dass die Deadline/Termine der Aufgaben theoretisch erhalten können.

2.1 Modell und Notation
Die Typen der Echtzeitsystemen sind umfangreich. Nach bestimmten Kriterien und anwendungsorienten Anforderungen funktionieren die Echtzeitsystemen ziemlich unterschiedlich. Es ist sehr schwierig über meiste „*real-life*" Ablaufplanungen zusammenzuurteilen, wir brauchen Abstraktion und Zuordnung.

In diesem Artikel werden wir nur über *harten* Echteitsystemen diskutieren. Das Wort „hart" bedeutet, dass bei Versäumen eines vorgegebenen Termins eine „Katastrophe" hervorrufen werden kann, nämlich Terminverletzung ist keinesfalls tolerierbar.

In unserem System nehmen wir an, dass *Aufgaben* (engl. Tasks) *Verdrängung* (engl. preemption) erleiden können. Zum Beipiel in *prioritätsorientierter Einplanung* (engl. priority-driven scheduling) ein hoch-priore Aufgabe kann einen nieder-priore Aufgabe verdrängt, wenn die zwei Aufgaben zur Zeit keiner Betriebsmittelkonflikt gibt. Hoch-priore Aufgabe kann auch von nieder-priore Aufgabe blockieren, wenn die nieder-priore Aufgabe ein Betriebsmittel, das hoch-priore Aufgabe auch braucht, schon vorher besessen hat.

Unter steht einige Erläuterungen für die Notation, die später im Artikel ständig benutzen werden.

Notation: Jede periodische Aufgabe A_i beteht aus eine Abfolge von *Arbeitsaufträgen* $J_{i,k}$ (engl. Jobs). Zu jeder Zeit sind die Periode und Ausführungszeit aller periodischen Aufgaben bekannt gegeben durch *a priori Wissen*. Notation $J_{i,k}$ bedeutet, diese Arbeitsauftrag ist die k^{th} Auftrag von A_i. Wenn alle Arbeitsaufträge von A_i gleiche Verhaltens haben, können wir einfach die Notation J_i benutzen.

Notation: Wir benutzen Notation $P(J_i)$ um die Priorität des Arbeitsauftrages J_i darzustellen. Und wir nehmen so an, dass in unserem Modell Prioritäten nach $P(J_1) > P(J_2) > P(J_3) ... > P(J_n)$.

Notation: Wir zeichnen *Auslastung* U_i (engl. utilization) einer Aufgabe J_i als $U_i = E_i / C_i$. Wir definieren totale Auslastung als U_{sum}. C_i ist die Periode von J_i, E_i ist die Ausführungszeit von J_i.

Notation: Wie benutzen *Locker/Semaphore* Mechanismus, um Betriebsmittelkonkurrenz zu löschen. Notation Ri stellt einen binären Semaphore für ein festgelegtes Betriebsmittel dar.

Notation: Wenn Job J_i ein Betriebsmittel R_i besessen hat, wir sagen es J_i ist jetzt im *kritischen Abschnitt* (engl. critical section), und wir zeichnen dieses Konzept als $Z_{i,j}$.

2.2 Gebräuchliche Schedulingsverfahren
Im Allgemeinen hat man meiste Schedulingsverfahren zu drei Kategorien aufgeteilt. Die erste Kategorie nennen wir

taktgesteuerte Verfahren (engl. clock-driven, auch time-driven). Taktsteuerung heißt, die Zeitpunkte von Einplanungsentscheidung sind typischerweise festgelegt, z. B. Durch Tabellen, die die Startzeitpunkte der einzelnen Arbeitsauftrage enthalten. Die Ablaufpläne werden vorgerechnet, nämlich auf Englisch *off-line scheduling*. Alle Verhaltens der Aufgaben sind vorhersehbar.

Wir nennen die zweite Kategorie als *vorranggesteuerte Verfahren* (engl. priority-driven, auch event-driven). Laut solchen Verfahren können die Aufgaben zufällig auftreten und somit ist das Systemsverhalten nicht mehr vorhersehbar. Aufgaben haben *beliebige* Auslösezeitpunkten (nach dem Ereigniszeitpunkt) und eigene Prioritäten. Der Scheduler des Systems steuert die *Einlastung* (engl. dispatching) der Aufgaben nach ihren Prioritäten. Arbeitsaufträge mit höherer Priorität haben Vorrang auf den Prozessor zu laufen. Prioritäten der Aufgaben werden off-line vergeben und ggf. on-line fortgeschrieben.

Letzte Kategorie ist bekanntes *reihum gewichtete Verfahren* (engl. Weight round-robin). Ein Arbeitsauftrag kann nur während eines bestimmten Zeitraums ausgeführt werden. Die Länge des Zeitraums passt sich zu Gewicht des Arbeitsauftrages ein. Wenn Seine Zeit abgelaufen ist, wird dieser Arbeitsauftrag in einer *Warteschlange* (engl. queue) eingefügt. Die Warteschlange arbeitet normalerweise nach *First-In-Frist-Out* Prinzip.

2.3 Zugriffskontrolle

Eine zentrale Problem vom Ablaufplanungsentwurf ist die Synchronisation zwischen mehreren Aufgaben. Die Ausführung einer Aufgabe nennen wir einen *Prozess*. Konkurrenz wird auftauchen, wenn mehrere Prozesse auf gemeinsame unteilbare Betriebsmittels nebenläufig zugreifen.

Was wird passiert? wenn wir keine Zugriffskontrolle verwenden. *Prioritätenumkehr* (engl. priority inversion) kann möglicherweise auftauchen, wenn unseres System Vorrangsteuerung verwendet. Stellen wir uns vor, ein nieder-priore Arbeitsauftrag J_n hat ein unteilbares Betriebsmittel R_i zuerst belegt. Dann versucht ein hoch-priore Arbeitsauftrag J_h dasselbe Betriebsmittel R_i zu belegen, aber R_i ist schon belegt, deshalb muss J_h warten bis das Betriebsmittel R_i wieder frei. Wir beschreiben oben vorgestellte Situation so im Echtzeitsystem, der nieder-priore Arbeitsauftrag hat hoch-priore Arbeitsauftrag blockiert und Prioritätsumkehr ist genau eine unerwünschte Folge von solcher Blockierung. Viel schlimmer ist, ein oder mehrere mittel-priore Arbeitsaufträge J_m können auch den nieder-priore Arbeitsauftrag J_n verdrängt. Jetzt muss wartender hoch-priore Arbeitsauftrag J_h auch auf mittel-priore Arbeitsaufträge warten, wenn ein mittel-priore Arbeitsauftrag das Betriebsmittel R_i nicht braucht, J_m kann sogar komplett ausgeführt werden. Wir nennen es *unkontrollierten Prioritätenumkehr*. Um unkontrollierten Prioritätenumkehr zu beseitigen, kann man *Prioritätsvererbung* (engl. priority inheritance) verwenden. Laut dieses Verfahrens wird ein nieder-priore Arbeitsauftrag die Priorität von hoch-priore Arbeitsauftrag vererbet, damit dürfen die zwischen mittel-priore Arbeitsaufträge nun diesen nieder-priore Arbeitsauftrag nicht mehr verdrängen. Neben Prioritätenumkehr können die Arbeitsaufträge sich auch *verklemmen*. Wenn das System geschachtelten Betriebsmittelzugriff erlaubt, zwei Arbeitsaufträge, die schon Betriebsmittels belegt haben, vielleicht verlangen weitere Betriebsmittels, z. B. Arbeitsauftrag J_1 belegt das Betriebsmittel R_1 und Arbeitsauftrag J_2 belegt das Betriebsmittel R_2. Und weiter J_1 verlangt R_2 und J_2 verlangt R_1. Aber R_1 und R_2 sind schon belegt, die zwei Arbeitsträge werden das schon belegte Betriebsmittel nicht *freiwillig* freigeben, deshalb können die Beiden nicht mehr weiter laufen, wir bezeichnen solche Situation als *Verklemmung* (engl. deadlock bzw. lifelock). Prioritätsvererbung ist nicht mehr vernünftig gegen Verklemmung. Bessere Gegenmaßnahme heißt *Prioritätsobergrenzen* (engl. priority ceiling protocol, PCP).

Da das *Multiprozessor-Priorität-Ceiling-Protokoll* Abk. MPCP den Schwerpunkt dieses Artikels ist und MPCP ist eigentlich eine Verbreiterung vom normalen PCP. Deswegen lassen wir das Prioritätsobergrenzen-Protokoll ein bißchen anschauen.

Laut des PCPs wird jedes Betriebsmittel im System eine Prioritätsobergrenze bestimmt. Wenn mehrere Arbeitsaufträge dasselbe Betriebsmittel belegen möchten, ist diese Prioritätsobergrenze gleich die Priorität des höchst-prioren Arbeitsauftrags. Wenn ein nieder-priore Arbeitsauftrag belegt ein Betriebsmittel, er kann die Prioritätsobergrenze vom diesen Betriebsmittel erben. Momentan lauft das PCP gleich wie Prioritätsvererbung, alle unkontrollierte Prioritätenumkehr wird ausgeschlossen. Jetzt lassen wir uns noch einmal oben dargestellte Deadlock-Szenarios durcharbeiten. J_1 und J_2 brauchen R_1 und R_2, das heißt die Prioritätsobergrenzen von beiden Betriebsmittels sind gleich $P(J_1)$, weil wir im System immer nach $P(J_1) > P(J_2) > \ldots P(J_n)$ annehmen. Bitte stellen wir uns vor, dieses Mal J_2 hat R_2 zuerst belget, dann wird J_1 gestartet, er wird J_2 verdrängt. Zu einem Zeitpunkt versucht J_1 das Betriebsmittel R_1 zu belegen. Laut des PCPs muss J_1 höhere Priorität als die *aktuelle Prioritätsobergrenze* besitzen, dann kann er das Betriebsmittel belegen. Die aktuelle Prioritätsobergrenze des Systems gleicht die höchste Prioritätsobergrenze von zu der Zeit schon belegten Betriebsmittels. Aus dieser Definition wissen wir sofort, die aktuelle Prioritätsobergrenze des Systems ist nun gleich die Prioritätsobergrenze von R_2, die genau gleich $P(J_1)$ ist. Deshalb ist die Priorität von J_1 nicht größer als die aktuelle Prioritätsobergrenze, also J_1 darf R_1 nicht belegen. J_2 wird weiter ausgeführt und später zu einem Zeitpunk versucht J_2 das Betriebsmittel R_1 zu belegen, natürlich gibt es hier kein Problem. Dadurch haben wir Deadlocks vorbeugen. Und die Idee hinter dem PCP ist um eine lineare Ordnung von Prioritätsobergrenzen des Sysrtems zu erhalten.

3. ECHTZEIT-SCHEDULING MIT MULTIPROZESSOREN

Die Ablaufplanungen bei Uniprozessorsystem sind ausgereift, nun können wir in eine andere Welt eintreten. Wie kriegt man gültige Ablaufplanungen in Multicoresystemen.

3.1 Neue Anforderungen.

Das erste auftauchende Problem in Multiprozessorsystemen ist *Zuordnungsproblem* (engl. tasks assignment). Normalerweise arbeiten die meisten Echtzeitsystemen statisch. Das heißt, Aufgaben sind mit zugeteilten Prozessoren gebunden. Die *Partitionen* der Aufgaben sind off-line entworfen und festgelegt. Falls wir nun in Multicoresystemen arbeiten, sollen wir geeignete Algorithmen finden, mit den alle Aufgaben und Betriebsmittels zu einigen Module aufgeteilt werden. Schließlich wird jeder Modul mit einem passenden Prozessor zugeordnet.

Zweite auftauchende Problem ist *Interprozessor Synchronization.*, nämlich die Synchronisation zwischen Prozessoren. Nach der Phase der Aufgabenzuordnung, meiste "*großen*" Aufgaben, die auf mehreren Prozessoren ausführen müssen ggf. mehreren Betriebsmittels brauchen, werden nach bestimmten Kriterien in "*kleinen Stücke*" nämlich *Unteraufgaben* (engl. subtasks) durchgeteilt. Dies folgen direkt Synchronizationsproblem zwischen jeder Unteraufgabe mit ihren *Nachbar-Unteraufgaben*. Hier nehmen wir so an, dass Aufgabe Ai kausal ist. Das heißt, alle Unteraufgaben, die Ai gehören, müssen nach einer festgelegten Reihenfolge ausgeführt werden. Dazu braucht man Protokolle für Synchronisation, um die Restriktion des Vortrittes zwischen Unteraufgaben zu erhalten.

Ein ziemlich spannender Punkt bei der Synchronisation zwischen Unteraufgaben ist die Bestimmung relativer Deadline für jede Unteraufgabe. Das Prinzip hier ist, solange die Deadline der ganzen Aufgabe und die Kausalität der Unteraufgaben nicht verletzt weden, man hat die Freiheit beliebige relativen Deadline für jede Unteraufgabe auszuwählen.

3.2 Aufgabenzuordnung

Meiste Hart-Echtzeitsysteme sind statisch, alle Aufgaben werden off-line zu Module partitioniert und jeder Modul wird mit eigenem Prozessor zugeteilt.

Eine einfache Zuordnungsmethod heißt *Simple Bin-Packing* [6]. In diesem Algorithmus sorgen wir nicht über die Kommunikationskosten zwischen Prozessoren, sondern wir versuchen nur minimale Anzahl der Prozessoren zu kriegen. Wenn totale Auslastung U_{sum} des Systems größer als 1, es bedeutet offensichtlich, we brauchen mehr als ein Prozessor in System. Jeder Arbeitsauftrag hat seine eigene Auslastung U_i. Wir wählen Arbeitsaufträge aus so viele wie möglich und die summierende Auslastung von solcher Arbeitsaufträge ist kleiner als 1. Dadurch haben wir einen Modul bestimmt und dieser Modul wird mit einem freien Prozessor zugeordnet. Man nennt diesen Modul als *a BIN* auf Englisch. Die Anzahl aller BINs, die wir brauchen, ist genau die Anzahl der nötigen Prozessoren des Systems.

Man muss hier beachten, um eine optimale Aufgabenzuordnung zu finden, ist eine *NP-harte* Problem [7]. Deswegen optimale Lösung gibt es selten, normalerweise verwendet man in der Praxis heuristische Verfahren, um teilweise optimale Zuordnung zu erreichen.

3.3 Multiprocessor-Priority-Ceiling-Protokoll

3.3.1 Spezifikation des MPCPs

Multiprocessor-Priority-Ceiliing-Protokoll nimmt an, dass Aufgaben und Betriebsmittels statisch mit dem zugeteilten Prozessor gebunden sind. Und alle Parameters der Aufgaben sind vorher schon bekannt. Das heißt der *Ablaufplaner/Scheduler* weißt die Prioritäten aller Aufgaben und welche Aufgaben brauchen welchen globale- und lokale Betriebsmittels. Wenn eine Aufgabe mit einem Prozessor zugeteilt wird, sie gehört immer zu diesem Prozessor und zwar nennen wir diesem Prozessor als *Host Prozessor* für diese Aufgabe.

Das MPCP nutzt binäre Semaphores als Synchronisationsmittel zwischen Aufgaben. Wenn ein Semaphore von mehreren Aufgaben zwischen mehreren Prozessoren zugegriffen wird, wir nennen es als *globaler Semaphore*. Die kritische Abschnitt, die von diesem Semaphore beschützt wird, nennen wir es *globale kritische Abschnitt* (engl. global critical section, gcs). Ein Semaphore heißt *lokaler Semphore*, wenn der Zugriff von anderen Aufgaben von anderen Prozessoren verboten ist. Und die kritische Abschnitt nennen wir *lokale kritische Abschnitt* (engl. local critical section, lcs). Laut des MPCPs wird ein Job seine lokalen kritischen Abschnitte und seine nicht-kritischen Abschnitte auf eigenen/lokalen Prozessor ausgeführt. Der Job laufet seine globalen kritischen Abschnitte auf *fernen* Prozessoren (engl. remote processor). Wir nennen die Prozessoren, die globalen Semaphores besitzen, *Synchronisationsprozessor* (engl. synchronisation processor).

Ein Job versucht einen globale Semaphore zu belegen. Wir nennen diese Aktion *Auswandern* (engl. migration). Wenn ein Job zur Zeit in seiner globalen kritischen Abschnitt laufet, das heißt, der laufet nun auf einen fernen Prozessor, deswegen seiner Host Prozessor ist frei, andere nieder-priore Jobs können auf den laufen. Im Uniprozessorsystem können die hoch-priore Jobs nur von nieder-priore Jobs blockiert werden. Aber im Multiprozessorsystem kann ein hoch-priore Job von irgend-priore Jobs blockiert werden, wenn dieser hoch-priore Job in seine gcs eintreten möchte. Um solche (remote) blockierende Zeit zu minimieren ist ein sehr wichtiges Ziel des MPCPs.

Wenn die Entwicklern die Aufgaben mit passenden Prioritäten zuordnen, sie müssen an einer zusätzlichen Anforderung denken. Gemäß des MPCPs hat jeder Prozessor einen eigenen Ablaufplaner, der steuert alle lokale Aufgaben und globale kritischen Abschnitte auf diesen Prozessor nach festgelegter Prioritätseinplanung und kontrolliert Betriebsmittelzugriff nach normalem PCP. Prioritätsobergenze (engl. priority ceiling) eines Betriebsmittels R_i ist die höchste Priorität aller Arbeitsaufträge/Jobs, die R_i benötigen. Wenn ein Job J_g zur Zeit auf einen Synchronisationsprozessor laufet, nämlich J_g ist momentan in seinem globalen kritischen Abschnitt. Und J_g hat globales Betriebsmittel R_g beleget. Dashalb $P(J_g)$ ist nun gleich die Prioritätsobergenze von R_g. Dann stellen wir vor, zu diesem Zeitpunkt ist ein lokaler Job J_l auf den Synchronisationsprozessor bereit, und J_l besitzt eine höhere Priorität $P(J_l)$ als J_g. Also J_l wird J_g verdrängt. Diese Situation sollen wir verhindern. Die Lösung ist einfach, jeder Job, der in seiner globalen kritischen Abschnitt laufet, besitzt eine höhere Priorität als irgenden lokalen Jobs, die diesem Synchronisationsprozessor gehören [2].

3.3.2 Ein einfaches Beispiel : Wie funktioniert das MPCP überhaupt?

Die Abbildung 1 hat uns die Struktur des Systems für Beispiel gezeigt. Diese Multiprozessorsystem besteht aus drei Prozessoren P_1, P_2 und P_3. Prozessor P_1 besitzt ein CPU und zwei lokale Betriebsmittels R_1 und R_2. Prozessor P_2 besitzt auch ein CPU zwar kein Betriebsmittel. Prozessor P_3 besitzt ein DSP (engl. digital signal processor) und ein globales Betriebsmittel R_3.

Arbeitsaufträge J_1 und J_2 sind mit P_1 zugeordnet. Die Beide benötigen die Betriebsmittels R_1 und R_2, und zwar J_2 braucht weiter das ferne Betriebsmittel R_3. Arbeitsaufträge J_3 und J_5 sind mit P_2 zugeordnet. J_3 benötigt das globale Betriebsmittel R_3. J_5 braucht kein Betriebnsmittel. Prozessor P_3 hat nur einen Arbeitsauftrag J_4 und ein globales Betriebsmittel R_3. J_4 verlangt nur R_3. Und $P(J_1) > P(J_2) > P(J_3) > P(J_4) > P(J_5)$

Auf unteren drei Abbildungen 2-4 können wir eine Senkrechte per Job sehen, die zeiget die Auslösezeit des Jobs. Weiße Rechtecke zeichnen Ausführen des Jobs ohne Belegung des Betriebsmittels. Rechtecke mit eigenen Farbe zeichnen Ausführungen der Arbeitsträgen mit Belegung des passenden Betriebsmittels.

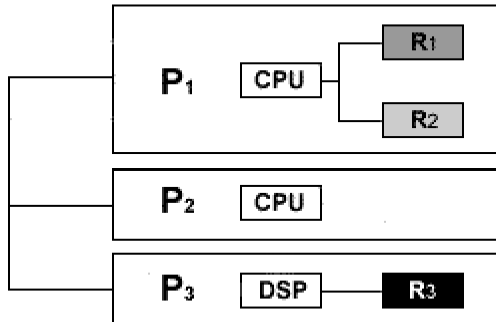

Abbildung 1: Das System Modell

Eine wichtige Bemerkung ist, dass wir annehmen, momentan im System ist geschachtelter Betriebsmittelzugriff zwischen Prozessoren noch nicht erlaubt.

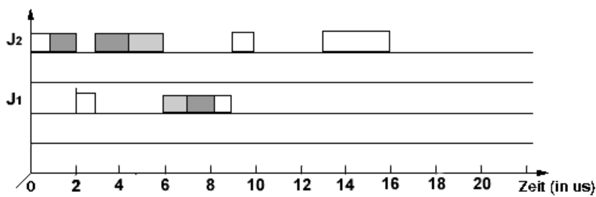

Abbildung 2: Schedulingsvorgang auf den Prozessor P_1

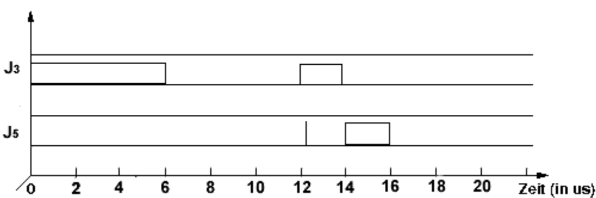

Abbildung 3: Schedulingsvorgang auf den Prozessor P_2

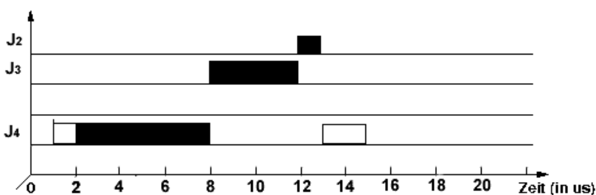

Abbildung 4: Schedulingsvorgang auf den Prozessor P_3

Fangen wir mit dem Schedulingsvorgang auf den P_1 an, hier werden wir nur die wichtigen Entscheidungspunkte ausgewählt.

1. Arbeitsauftrag/Job/Thread J_2 wird zum Zeitpunkt T_0 gestartet und zum Zeitpunkt $T_{0.9}$ belegt das Betriebsmittel R_1, J_2 ist nun in seinem lokalen kritischen Abschnitt.

2. Zum Zeitpunkt T_2 wird J_1 gestartet und er verdrängt den nieder-prioren Thread J_2.

3. Zum Zeitpunkt T_3 versucht J_1 das Betriebsmittel R_2 zu belegen. Weil wir für lokalen Betriebsmittelzugriff das normale (lokale) PCP verwenden, das ist völlig klar, dass J_1 blockiert wird, weil J_2 R_1 schon belegt und dessen Prioritätsobergrenze erbt. Die Priorität von J_2 ist gleich wie J_1. Deswegen wird J_2 weiter ausgeführt. Zur Zeitpunkt T_4 belegt J_2 auch R_2. Schließlich wird J_2 zum Zeitpunkt T_6 alle lokalen Betriebsmittels wieder freigegeben, und zwar J_2 wird auf seine ursprüngliche Priorität zurückfällt. Jetzt kann J_1 endlich R_2 belegen und weiter laufen bis zum Schluss. Zwischen diesem Zeitraum wird J_2 verdrängt.

4. Zum Zeitpunkt T_9 wird J_2 weiter ausgeführt. Zum Zeitpunkt T_{10} versucht J_2 das ferne Betriebsmittel R_3 zu belegen. Bevor wir zu Prozessor P_3 wandern, lassen wir mal gucken, was hat auf den Prozessor P_2 passiert. Zum Zeitpunkt T_0 wird J_3 gestartet und zum T_6 versucht J_3 das ferne Betriebsmittel R_3 zu belegen. Er hat seiner lokale Prozessor P_2 verlassen.

5. Auf den Prozessor P_3 ist der Schedulingsvorgang ein bisschen kompliziert. Zum T_1 wird J_4 getstartet und zum T_2 belegt J_4 das Betriebsmittel R_3 bis T_8. Das heißt, zum Zeit T_6 darf J_3 (kommt aus P_2) das Betriebsmittel R_3 nicht belegen. J_3 wartet bis T_8.

6. Zum Zeitpunkt T_8 gibt J_4 R_3 frei, J_3 kann nun auf P_3 laufen. Wir nennen J_3 ist jetzt in seiner globalen kritischen Abschnitt. Interessant ist zum Zeitpunkt T_{10} ist J_2 aus P_1 angekommen, der auch R_3 verlangt, aber R_3 ist zur Zeit von J_3 belegt. Jetzt der Scheduler von P_3 wird nach einem (globalen) PCP J_2 und J_3 steuern. Das heißt R_3 besitzt zwei Prioritätsobergrenzen, eine für Zugriffskontrolle von lokalen Arbeitsaufträgen und eine für Zugriffskontrolle von *ankommenden Arbeitsaufträge*. J_3 erbt *globale Prioritätsobergrenze* von R_3, J_2 muss darauf warten.

7. Zum Zeitpunkt T_{12} gibt J_3 R_3 wieder und verläßt seine globale kritische Abschnitt. J_2 kann nun R_3 belegen und in seine globale kritische Abschnitt eintreten.

3.3.3 Zeitanalyse

Von oben dargestellter Beschreibung des MPCPs kann man ein paar interessante Punkte sammeln. Zum Beispiel wird der globale kritische Abschnitt von J_2 wegen J_3 während Zeitintervall (10, 12] verzögert. Wenn wir die Planbarkeit von J_2 analysieren, wir müssen solche Verzögerungen zusammenrechnen. Zum Zeitpunkt T_6 wird J_3 auf P_2 suspendiert. Es folget, bis zum Zeitpunkt T_{12} ist J_3 immer nicht fertig, und J_5 ist zum Zeitpunkt $T_{12.1}$ schon bereit, aber wird von J_3 verzögert bis zum Zeitpunkt T_{14}. Solche Blockierungen sind unerwünscht und haben große Auswirkung zur Verletzung der Performance des Systems. Laut Sha's Papier [2] leiden Jobs unterschliedliche Blockierungen von fünf Typen.

1) Lokale Blockierungszeit (engl. local blocking time). Wegen des Wettbewerbes um lokalen Betriebsmittels (engl. resource contention) mit anderen Jobs auf Host Prozessor

2) Lokale Verdrängungsverzögerung (engl. local preemption delay). Wegen der Verdrängung der Jobs aus anderen Prozessor.

3) Ferne Blockierungszeit (engl. remote blocking time). Wegen des Wettbewerbes um Betriebsmittel mit der nieder-prioren Jobs aus Synchronizationsprozessor.

4) Ferne Verdrängungsverzögerung (engl. remote preemption delay). Wegen der Verdrängung der hoch-prioren Jobs aus Synchronizationsprozessor

5) Aufschiebbare Blockierungszeit (engl. deferred blocking time). Wegen Suspendierung der hoch-prioren Jobs aus Host Prozessor.

Wir können die fünf Zeiteinheiten summieren, dadurch kriegen wir *totale Blockierungszeit B* (engl. total blocking time). B ist die Obergrenze der Blockierungszeit eines Jobs J_i. Weil wir hier das Szenario im ungünstigsten Fall (engl. Worst-case scenario) angenommen haben. Im Alltags ist es ziemlich selten, wenn ein Job alle fünf Blockierungen einmal eingetroffen hat.

3.3.4 Die Einwirkung geschachteltes Betriebsmittelzugriffes

Wenn geschachtelter Betriebsmittelzugriff zwischen Prozessoren ist erlaubt, eine ganz offensichtliche Wirkung ist die totale Blockierungszeit wird verlängern, weil die Möglichkeit, dass ein Job blockiert wird, vergrößert. Das ist noch nicht so schlimm. Mehr gefährliche Auswirkung ist, dass die Verklemmungen können wieder auftauchen.

Von Abschnitt 2.3 Zugriffskontrolle wissen wir schon die Folge, wenn man den geschachtelten Betriebsmittelzugriff (engl. nested requests for resources) im Uniprozessorsystem erlaubt, Verklemmungen können zwischen Threads auftauchen. Dazu ist die Prioritätsobergenzen eine passende Lösung. Durch PCP haben wir die Verklemmungen verhindert. Aber wir werden sofort sehen, das ist nicht weiter gültig im Multiprozessorsystem, wenn folgendes Szenario passiert.

Stellen wir uns vor, J_1 und J_2 sind mit P_1 gebunden, R_1 ist das lokale Betriebsmittel auf den P_1. Im System gibt es auch einen Prozessor P_2 mit einem globale Betriebsmittel R_2, J_1 und J_2 beide benötigen R_1 und R_2. Wir nehmen so an, J_2 wird zuesrt gestartet auf P_1 und zu einem Zeitpunkt belegt R_1. Dann wird J_1 gestartet und J_2 wird verdrängt, Weil die Priorität von J_1 größer als J_2. Auch zu einem Zeitpunkt J_1 versucht R_2 auf den P_2 zu belegen, weil R_2 zur Zeit noch frei ist, wird J_1 R_2 kriegen und weiter auf P_2 ausgeführt. Auf P_1 kann J_2 nun wieder laufen. Wir sollen uns noch erinnern, der Betriebsmittelzugriff ist geschachtelt, das heißt, zu einem Zeitpunkt vielleicht J_2 braucht R_2 und J_1 bracuht R_1, aber das andere Betriebsmittel ist immer belegt, also Beide werden blockiert, nämlich die verklemmen sich.

Wenn wir den geschachtelten Betriebsmittelzugriff zwischen Prozessorn nicht erlaubt, dann brauchen wir neues Modell.

4. ABLAUFPLANUNG FÜR END-TO-END AUFGABEN MODELL

4.1.1 End-to-End Aufgaben Modell

Wenn geschachtelter Betriebsmittelzugriff verboten ist, wir zeichnen eine Aufgabe, die mehr als ein Prozessor (Betriebsmittel) nötigen, als eine End-to-End Aufgabe. Eine Aufgabe A_i besteht aus ein paar Komponenten. Jede Komponente stellt einen Arbeitsauftrag $J_{i,j}$ dar, der nur auf einen Prozessor ausgeführt wird. Weil alle Arbeitsaufträge einer Aufgabe die Kausalität erhalten müssen, es heißt ein folgender Arbeitsauftrage $J_{i,j}$ darf nicht starten, solange seiner Vorgänger $J_{i,j-1}$ noch nicht fertig ist. Deswegen ist die Reihenfolge aller Komponenten/ Arbeitsaufträgen einer Aufgabe festgelegt. Normalerweise solche Arbeitsaufträge werden eine Warteschlange-Datenstruktur aufbauen.

Ein Beispiel: Aufgabe A_i hat lokale und globale kritische Abschnitte. A_i braucht ein lokales Betriebsmittel zuerst, dann ein globales Betriebsmittel , am Schluss wieder ein lokales Betriebsmittel. Wenn es keinen geschachtelten Betriebsmittelzugriff zwischen Prozessoren gibt, A_i kann als eine End-to-End Aufgabe betrachten, die aus drei Arbeitsaufträge $J_{i,1}$, $J_{i,2}$ und $J_{i,3}$ besteht.

Typischerweise wird ein End-to-End Schedulingsverfahren in zwei Teile aufgeteilt. Der erste Teil ist über die Synchronisation zwischen einem Arbeitsauftrag und seinem Vorgänger oder Nachfolger, der auf anderen Prozessor lauft. Der zweite Teil ist über gültiges und effektives Ablaufplanungsalgorithmus auszuwählen. Gemäß des MPCPs die Prioritäten des Arbeitsaufträge sind festgelegt, dagegen nun hat man die Freiheit einen passenden Schedulingsverfahren zu nutzen, zum Beispiel LRT (engl. latest release-time first). Man kann sogar vermischte Verfahren auf verschiedene Prozessoren verwenden.

4.1.2 End-to-End Synchronisationsprotokoll

Die zentrale Frage im End-to-End Synchronisationsprotokoll ist, wann darf ein Arbeitsauftrag nach den Abschluss seines Vorgängers ausgelöst. Kann er sofort los laufen oder muss auf anderen warten? Genau zu diesem Punkt werden meiste Synchronisationsprotokolle in zwei Kategorien aufgeteilt. Eine Kategorie heißt *gierige* (engl. greedy) Verfahren und andere heißt *ungrierige* (engl. non greedy) Verfahren. Nach gierigen Verfahren wird einer Arbeitsauftrag sofort starten, wenn seiner Vorgänger fertig ist. Nach ungierigen Verfahren kann der Arbeitsauftrag sofort losgehen oder auf anderen Arbeitsauftrag warten, der zwar andere Aufgabe gehört, aber auf denselben Prozessor ausgeführt wrid.

Mehr Details über End-to-End Synchronisationsprotokoll können Sie das Papier von Sun und Liu bekommen [8].

4.1.3 Schedulingsverfahren der Unteraufgaben auf jeden Prozessor

Für Unteraufgaben/Arbeitsaufträge, dieselbe Aufgabe gehören, ist die Prioritätszuordnung als eine zentrale Problem aufgetaucht. Natürlich ein Arbeitsauftrag darf eine unterschiedliche Priorität gegen seinem Vorgänger oder Nachfolger besitzt. Und Wir haben schon gelehrt, Prioritätszuordnung ist eine NP-harte Problem. in der Praxis benutzt man normalerweise heuristische Verfahren. Solche Verfahren arbeiten meistens in zwei Schritte. Zuerst werden relative Deadlines für Arbeitsaufträge berechnen. Solange die Frist der Aufgabe und die Kausalität der Arbeitsaufträge nicht verletzt wird, der Schedulingsentwerfer hat vollständige Freiheit die relative Frist für jeden Arbeitsauftrag zubestimmen. In der Papier von Kao und Garcia [4] können Sie ausführliche Beschreibung über *Fristszuordnung* (engl. deadline assignment) bekommen.

Nach oben dargestellter Phase sollen alle Unteraufgaben/ Arbeitsaufträge eigene Deadlines haben. Für einen bestimmte Prozessor ist die Arbeitsumgebung nun gleich wie in einem Uniprozessorsystem. Der Scheduler dieses Prozessors kann alle Arbeitsaufträge nach ihren relativen Fristen als lokalen Fristen einplanen, z. B. mit EDF-Verfahren (engl. earliest deadline first).

5. FAZIT

Dieser Artikel hat einen Überblick über die Ablaufplanung in Multiprozessorsystemen gegeben. Wenn unseres System statisch ist, nämlich die Informationen der Aufgaben sind a priori Wissen, ein typischer Schedulingsverfahren besteht aus grundsatzlich drei Schritte.

Der erste Schritt ist Aufgabenzuordnung. Hier wrid der Scheduler festgelegt, Welche Aufgaben oder Unteraufgaben auf welchen Prozessoren ausgeführt werden.

Der zweite Schritt ist Synchronisation der Arbeitsaufträgen zwischen Prozessoren. Hier soll man einen geeignete Verfahren auswählen, damit die Kausalität aller Arbeitsaufträgen erhalten können, z. B. Die Datenabhänigkeit darf nicht verletzt.

Der dritte Schritt ist Schedulngsverfahren. In diesem Artikel haben wir zwei Algorthmen gesehen, MPCP und Scheduling für End-to-End Aufgaben Modell. Wenn ein System auf MPCP basiert ist, jeder Arbeitsauftrag besitzt eine feste Priorität. Dagegen haben die Systeme, die auf End-to-End Modell basiert sind, mehrere Freiheit um ein passendes Schedulingsverfarhren auszuwählen.

6. DANKSAGUNGEN

Vielen Dank für die Korrekturen und Hinweise von Betreuer Niko Böhm.

7. REFERENCES

[1] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. IEEE Transactions on Computers, 1990. Ding, W. and Marchionini, G. 1997 A Study on Video Browsing Strategies. Technical Report. University of Maryland at College Park.

[2] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In Proc. of the 9th IEEE Real-Time Systems Symposium. IEEE, 1998.

[3] Sun,J., "Fixed priority scheduling of end-to-end periodic tasks," Ph.D. theiss, Department of Computer Science, Univeristy Of Illinoist at Urbana-Champaign, 1997.

[4] Kao, B., and H. Garcia-Molina, "Deadline assigment in distributed soft real-time systems," Proceeding of 13th IEEE Internationl Conference on Distributed Computing Systems,1993.

[5] Jane W. S. Liu. Real-Time systems. Prentice Hall, ISBN 0-13-099651-3, 2000.

[6] Coffman Jr., E. G., Garey, M.R. and Johnson, D.S. Approximation Algorithms for Bin Packing – An Updated Survey. Bell Laboratories, Murray Hill, N. J., 1983.

[7] Garey, M. R., D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.

[8] Sun, J., and J. W. S. Liu, “End-to-end Synchronization protocols of fixed priority periodic tasks,” Proceedings of the 16th IEEE International Conference on Distributed Computing Systems, Hong Kong, June 1996

1. Wird die äußere Form der Arbeit dem Anspruch an wissenschaftliche Texte gerecht?

2. Ist das Dokument klar strukturiert und ein durchgängiger roter Faden zu erkennen?

3. Sind die einzelnen Gliederungspunkte klar und verständlich dargelegt?

4. Werden komplexe Zusammenhänge genügend klar veranschaulicht und auf möglichst einfache Art und Weise dargestellt?

5. Wie ist der Gesamteindruck? Hast du verstanden, was der Verfasser dir vermitteln will?