Lock Inference

in

System Software

Stefanie Mika

Gliederung:

1. **Motivation**

2. **Lock inference**

3. **TSL**

4. **Beispiel**

5. **Zusammenfassung**

6. **Bibliographie**

1. Motivation:

In vielen Systemen spielt Nebenläufigkeit eine große Rolle, durch sie entstehen kritische Abschnitte, wenn zwei oder mehr Tasks gleichzeitig auf eine Ressource zugreifen wollen, auf die aber zu einem Zeitpunkt nur ein Task zugreifen darf. Deshalb ist es wichtig den Zugriff auf diese Abschnitte durch Sperren zu regeln. Doch ist die Implementierung dieser Locks nicht gerade einfach. Sie wird immer durch die Anforderungen des jeweiligen Systems bestimmt, schließlich beeinflussen Locks je nach Implementierung Antwortzeit und Durchsatz mehr oder weniger. Deshalb sind Sperren in Echtzeitsystemen zum Beispiel anders zu implementieren als in anderen Systemen. Nun könnte man zur Bauzeit des Systems eine Analyse des gesamten Programms vornehmen und anhand dieser Analyse über die Implementierung der Locks entscheiden. Dabei sind die komplexen Beziehungen zwischen mehreren Umgebungen und verschiedenen Kontexten zu beachten. Als Nebeneffekt der Herausarbeitung und Überprüfung der Synchronisation wird Lock Inference möglich.

2. Lock Inference:

Lock Inference wird zur Herleitung einer angemessenen Implementierung für jeden kritischen Abschnitt eines Systems genutzt und löst einige Probleme bei der Schaffung von flexibler, zuverlässiger und effizienter Software. Man hofft, durch Lock Inference robuste und auch wieder verwendbare Systeme entwickeln zu können und sich die Entwickler zudem nicht mehr mit den komplexen Regeln der Lock-Implementierung auseinandersetzen müssen. Auch soll der Code übersichtlicher, leichter veränderbar und weniger bug-anfällig werden. Überflüssige Locks in einem System sollen erkannt und entfernt werden. Dazu wird schon zur Entstehungszeit eines Systems eine umfassende Analyse mit Hilfsmitteln wie der task scheduler logic (im Folgenden mit TSL bezeichnet) vorgenommen werden. Wenn nun durch die Analyse des Systems keine passende Implementierung gefunden werden kann, hat man einen oder mehrere Fehler oder race conditions festgestellt und kann das System so, wie es im

Moment ist, nicht realisieren. Auch können durch die Analyse die Sperren an die Voraussetzungen und Anforderungen des jeweiligen Systems angepasst werden.

3. TSL

TSL ist ein Formalismus um die „inneren Vorgänge“ von scheduling und Nebenläufigkeit zu beschreiben. Durch ihn lassen sich race conditions und andere Fehler feststellen. In TSL werden die Regeln, mit denen Locking in Systemsoftware gesteuert wird, beschrieben. Durch die Analyse mit TSL lassen sich Nebenläufigkeitsprobleme finden, auch bei mehreren Ausführungsumgebungen. Vorteil von TSL ist, dass sich aus der Analyse automatisch eine angemessene Implementierung der Locks ableiten lässt. Die Schlüsselidee hinter TSL ist hierarchisches Scheduling. Als Tasks werden dabei alle schedulbaren Einheiten bezeichnet und als Scheduler alles, das die Reihenfolge der Tasks steuert. Properties werden einem Task von seinem Scheduler zugewiesen und auch vererbt. Nun ist im hierarchischen Scheduling jeder Scheduler aus Sicht eines übergeordneten Schedulers wiederum ein Task. Die Fähigkeit bzw. Unfähigkeit einen anderen Task zu unterbrechen wird vererbt, d.h. wenn ein höherer Scheduler einen Task nicht unterbrechen kann, so kann dies auch keines seiner Kinder. TSL bezieht sich im Generellen auf Systemsoftware und trifft keine Annahmen über die zugrunde liegenden Komponenten, trotzdem ist es besonders nützlich um komponentenbasierte Systeme zu analysieren, da diese Systeme dazu tendieren Interfaces für Locking bereitzustellen und es so leichter machen Synchronisation zu analysieren und zu parametrisieren. Obwohl sie aufgrund der vielen indirekten Verbindungen zwischen den einzelnen Modulen ziemlich komplex und schwer verständlich sind. TSL braucht für die Analyse eine statische Identifikation von Tasks, Schedulern, Ressourcen, kritischen Abschnitten und einen Callgraph des Programms oder Systems. Um ein System in TSL zu beschreiben, werden eigene Symbole verwendet. Man schreibt $t_1 \ll t_2$ wenn ein Task t_2 einen anderen Task t_1 verdrängen kann. Falls ein Task einen anderen verdrängen kann auch wenn dieser Locks L hält, wird dies durch $t_1 \ll_L t_2$ ausgedrückt. Wenn ein t_1 Scheduler von t_2 ist, schreibt man $t_1 \blacktriangleleft t_2$. $t \rightarrow r$ bedeutet nun, dass ein Task t eine Ressource r verwendet. Falls t Locks L hält, während er r verwendet, wird $t \rightarrow_L r$ geschrieben. Eine race condition kann nun auftreten, wenn zwei verschiedene Tasks t_1 und t_2 eine gemeinsame Ressource benutzen, sie

eine gemeinsame Anzahl von Locks $L_1 \cap L_2$ haben und t_2 t_1 unterbrechen kann, auch wenn dieser die Locks gerade hält.

$$\begin{aligned} race(t_1, t_2, r) = {} & t_1 \rightarrow_{L1} r \ \wedge \\ & t_2 \rightarrow_{L2} r \ \wedge \\ & t_1 \neq t_2 \quad \wedge \\ & t_1 \ll_{L1 \cap L2} t_2 \end{aligned}$$

Die Sperren werden vom jeweiligen Scheduler bereitgestellt (t -◦ l). Es wird davon ausgegangen, dass ein Lock nur von einem Scheduler verwaltet wird. Dabei gibt es zwei Arten von Locks, solche die jeden Task sperren, die versuchen auf die jeweilig Ressource zuzugreifen, und diejenigen, die nur die Tasks sperren, die die gleichen Locks haben. Wenn nun ein Task einen Lock anfordert, kann es sein, dass er blockiert werden muss, wenn die Ressource nicht frei ist. Dann blockiert der Scheduler, der die jeweilige Sperre bereitstellt, sein gerade laufendes Kinder, das ein Vorfahr des Tasks sein muss, der die Sperre angefordert hat, und blockiert somit auch diesen. Falls kein Vorfahr des Tasks den geforderten Lock verwaltet, hat man eine ungültige Aktion festgestellt. So eine Aktion wird mit *illegal*(t, l) bezeichnet. Die Zuweisung einer Sperre ist nur dann gültig, wenn der Lock nicht blockierend ist oder er von ein Vorfahr zur Verfügung gestellt wird. Zur Zeit zählt man noch solange alle gültigen Zuweisungen von Sperren zu kritischen Abschnitten auf bis man eine Zuweisung gefunden hat, die alle race conditions behebt. Falls keine derartige Zuweisung gefunden werden kann, wurde eine echte race condition entdeckt und das System kann so nicht realisiert werden. Bisher wird dieser Brute-force-Algorithmus verwendet, doch sucht man nach Verbesserungen. Man könnte zum Beispiel die Locks in einer intelligenten Reihenfolge überprüfen und nicht in einer Willkürlichen. Überflüssige Synchronisation wird durch die Synchronisationsableitung vorweggenommen. Es muss nur sichergestellt werden, dass zu jedem kritischen Abschnitt zumindest der so genannte „null lock“ verfügbar ist, dieser hat keinen Effekt auf Preemptionbeziehungen und wird als NOP implementiert. Wenn das System nun realisierbar ist, würde man natürlich am liebstem optimale Locks verwenden, doch darauf ist TSL nicht ausgelegt, denn es bevorzugt keine besondere Art von Sperren. Dieser Aspekt muss nun an anderer Stelle vorgenommen werden. Auch für Echtzeitbelange wird man andere Mittel als TSL verwenden, denn TSL kann nicht voraussagen, wie lange ein

Task eine Ressource wirklich belegen wird und ob ein Task eine Ressource länger blockieren wird, als er längstens dürfte.

4. Beispiel

Kommen wir nun zu einem kleinen Beispiel. Bild 1 zeigt die Software für eine eingebettete Applikation, die etwa eine Pumpstation an einer Ölpipeline überwachen könnte und Informationen über das System an HTTP-Klienten liefern könnte.

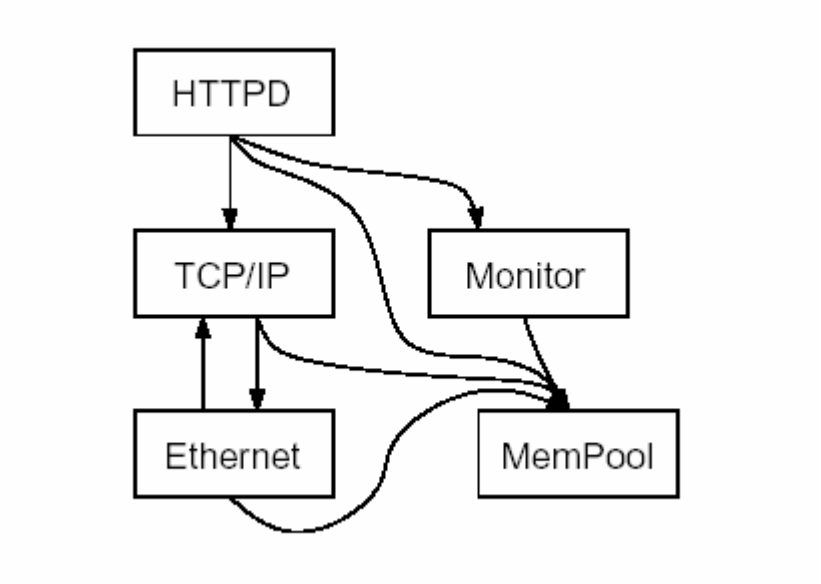

Bild 1

Auf Anfrage empfängt die HTTPD Komponente Daten von der Monitorkomponente und schickt sie mit Hilfe der TCP/IP- und der Ethernetkomponente aufs Netz. Alle Komponenten verwenden den Speicher. Bild 2 zeigt das System von Bild 1 etwas genauer. Links ist die Scheduling Hierarchie zu sehen, rechts die Komponenten.

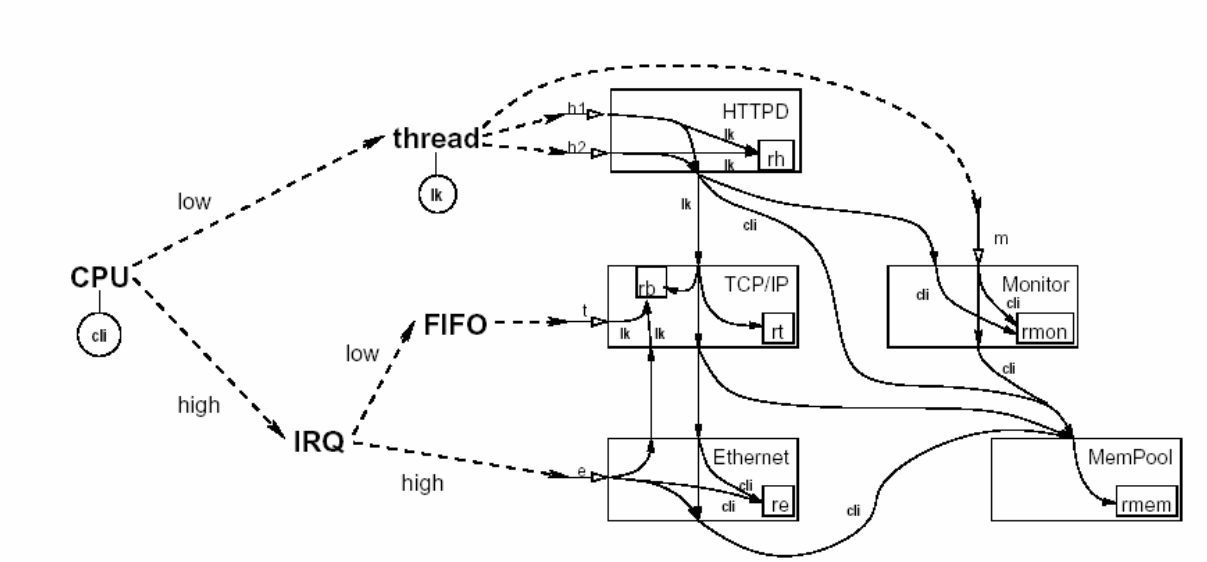

Bild 2

Bevor nun das System analysiert werden kann, müssen erst Bezeichnungen für die einzelnen Tasks (hier h1, h2, t, e und m), Scheduler (hier CPU, thread, FIFO und IRQ). Auch für die Ressourcen müssen Namen gefunden werden (hier rh, rt, re, rmon und rmem). Die Sperren (hier mit cli und lk bezeichnet) müssen hinzugefügt werden und sie den Schedulern zuweisen. Jetzt überprüfen wir, ob Fälle von ungültigen Sperren auftreten können. Dabei entsteht folgende Tabelle:

$h1, h2 \rightarrow_{lk}$ {rh, rt, rb, rmem}
$h1, h2 \rightarrow_{lk, cli}$ {re, rmem}
$h1, h2 \rightarrow_{cli}$ {rmon, rmem}
$m \rightarrow_{cli}$ {rmon, rmem}
$t \rightarrow_{lk}$ {rb}
$e \rightarrow_{lk}$ {rb}
$e \rightarrow_{cli}$ {re, rmem}

Aufgrund dieser Tabelle und dem Wissen, dass lk ein blockierender Lock ist, kann man eine Liste der Situationen bei denen es zu ungültigen Verwendungen der Locks kommt.

illegal(t, lk)
illegal(e, lk)

Ursache dieser beiden Probleme ist, dass der Lock lk verwendet wird, um die Ressource rb zu schützen, auf die durch Hardware- und Softwareinterrupts zugegriffen wird. Dies kann gelöst werden indem man anstelle von lk cli verwendet. Nun überlegen wir uns die Preemptionbeziehungen, dabei kommen wir auf folgende Tabelle.

h1, h2, m «ø h1, h2, m
h1, h2, m «ø t
t «ø e

Dies zusammen mit der Definition von race und der Nutzung der Ressourcen führt zu folgenden race conditions:

race(h1, h2, rmem) *race*(h2, h1, rmem)
race(h1, m, rmem) *race*(h2, m, rmem)
race(h1, e, rmem) *race*(h2, e, rmem)

Diese können aufgelöst werden, indem man den cli Lock verwendet, wenn man von TCP/IP MemPool aufruft. Das System hat keine überflüssigen Sperren mehr, wenn man jetzt aber nur einen single thread für die HTTPD-Komponente instanziieren könnte, würden die Locks, die rh schützen, überflüssig und könnten entfernt werden, genauso wie der Lock, der sicherstellt, dass auf die obere Hälfte der TCP/IP-Komponente nur atomar zugegriffen wird.

5. Zusammenfassung

Lock Inference kann die Arbeit als Entwickler von Softwaresystemen leichter machen, doch die Hoffnung, dass sich die Entwickler nicht mehr mit den Schwierigkeiten der Lockimplementierung auseinandersetzen müssen, wird wahrscheinlich genau das bleiben. Ganz wird Lock Inference oder Mittel wie TSL den Entwicklern diese Arbeit wahrscheinlich nie abnehmen. TSL ist bisher soweit, dass es, wenn man alle Vorgaben, wie Komponenten und Zugriffsgraph hat, zu guten Implementierungen führen kann, doch bei dynamischen Systemen kann man es noch nicht wirklich anwenden, nur für solche Komponenten, die nicht dynamisch veränderbar sind. TSL soll aber in dieser Richtung auch weiterentwickelt werden

und es gibt noch andere Hilfsmittel, die man zur Unterstützung von TSL für dynamische Systeme verwenden kann. Auch Deadlocks können bisher mit TSL noch nicht entdeckt werden, aber auch in dieser Hinsicht soll TSL noch verbessert werden. Man wird sehen, inwieweit sich TSL weiterentwickelt. Es ist sicher eine gute Idee und viele Entwickler werden froh sein, wenn die Implementierung von Locks einfacher werden würde.

6. Bibliographie

- Paper: „Lock Inference for system software“; John Regehr, Alastair Reid