Lock Inference in System Software

Stefanie Mika

Gliederung

- Motivation
- Probleme
- Lock Inference
- TSL
- Beispiel
- Zusammenfassung und Ausblick

Motivation

- Nebenläufigkeit in Systemen

  => kritische Abschnitte

- Angemessene Implementierung von Locks notwendig

Probleme

- Anforderungen des Systems
- Locks beeinflussen Performance
  - Notwendigkeit??
- Beziehungen zwischen einzelnen Komponenten
- Schwer zu implementieren

Lock Inference

- Herleitung einer angemessenen Implementierung der Locks
- Keine überflüssigen Locks mehr
- Entwickler müssen die Regeln des Lockings nicht mehr lernen
- Code verständlicher, weniger bug-anfällig

TSL

- Mittel zur Analyse eines Systems
- Entdeckt race conditions und andere Fehler
- Automatische Ableitung einer geeigneten Lockimplementierung
- benötigt zur Analyse statische Identifikation von Tasks, Schedulern, Ressourcen, kritischen Abschnitten und einen Callgraph

Grundlagen

- Hierarchisches Scheduling
- Task = schedulbare Einheit
- Scheduler = steuert Reihenfolge der Tasks
- Scheduler aus Sicht eines übergeordneten Schedulers selber Task
- Properties werden vererbt

Schreibweise

- $t_1 \ll t_2$ = t_2 kann t_1 verdrängen
- $t_1 \ll_L t_2$ = t_2 kann t_1 verdrängen während t_1 die Locks L hält
- t -> r = t verwendet Ressource r
- $t \rightarrow_L r$ = t hält Locks L wenn er Ressource r verwendet
- t_1 ◂ t_2 = t_1 schedult t_2

Race Condition

- 2 Tasks benutzen dieselbe Ressource
- gemeinsame Anzahl von Locks
- Task 2 kann Task 1 unterbrechen auch wenn dieser die Locks hält
- $race(t_1, t_2, r) = t_1 \rightarrow_{L_1} r$
  $\wedge\ t_2 \rightarrow_{L_2} r$
  $\wedge\ t_1 \neq t_2$
  $\wedge\ t_1 \ll_{L_1 \wedge L_2} t_2$

Lockanforderung

- Anfrage eine Tasks für einen Lock
- Suche nach Scheduler, der den Lock bereitstellt in der Hierarchie nach oben
- Falls keiner gefunden wird
  - ungültige Anforderung
- illegal(t, l)

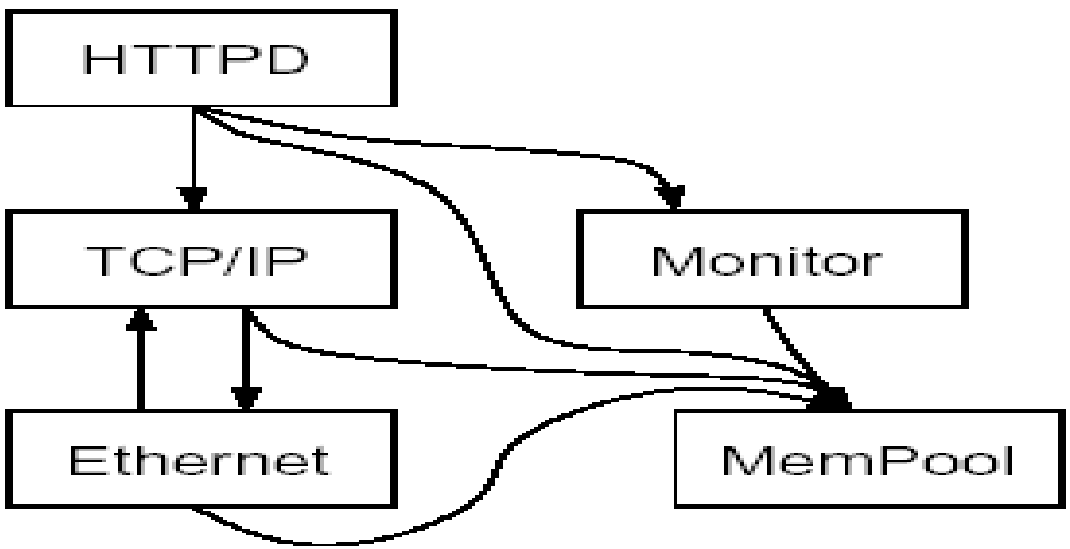
Algorithmus

- Bisher: Aufzählen aller gültigen Zuweisungen

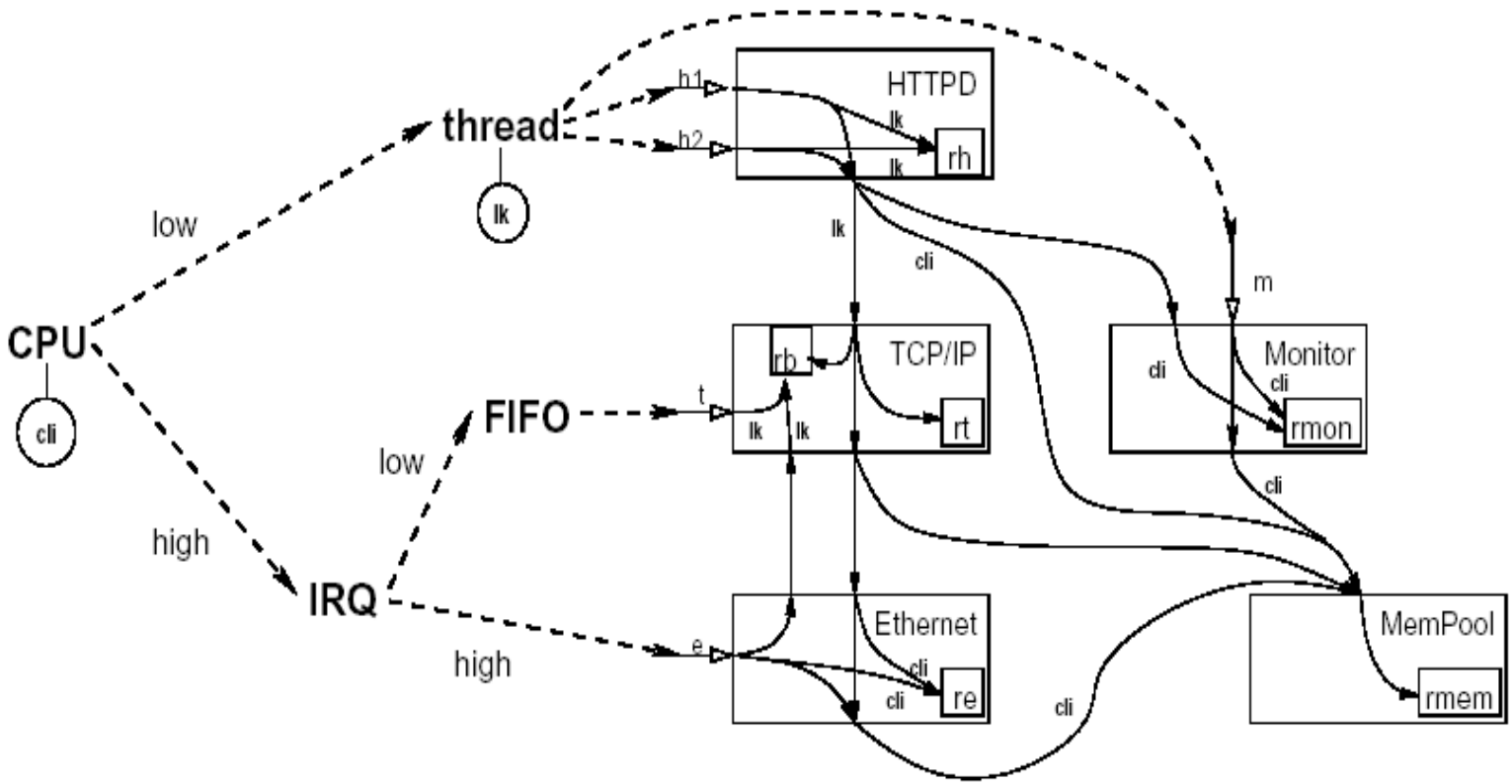
- Verbesserungen erwünscht
  - intelligentere Reihenfolge
  - Eigenschaften einer Domain ausnutzen

Beispiel

- Ein einfaches,eingebettetes Monitoringsystem



Detaillierterer Graph



Analyse

- Ressourcenzugriff
  - h1, h2 $\rightarrow_{lk}$ {rh, rt, rb, rmem}
  - h1, h2 $\rightarrow_{lk,\ cli}$ {re, rmem}
  - h1, h2 , m $\rightarrow_{cli}$ {rmon, rmem}
  - t, e $\rightarrow_{lk}$ {rb}
  - e $\rightarrow_{cli}$ {re, rmem}
- => illegal(t, lk), illegal(e, lk)

Preepmtionbeziehungen

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

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)

Zusammenfassung

- Verwendung von TSL ist Erleichterung für Entwickler
- Aber
  - Nur Information über Realisierbarkeit eines Systems nicht über Optimalität
  - TSL nicht für dynamische Systeme, da statische Definitionen gebraucht werden

Ausblick

- Prototyp eines TSL-Checkers
- Testumgebung basiert auf Knit (Komponentensprache) und OSKit (Library von Systemsoftware Komponenten
- TSL für dynamische Systeme