

Aufgabe 4: Thread Synchronization

Echtzeitsysteme - Übungen zur Vorlesung

Fabian Scheler, Peter Ulbrich, Niko Böhm

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

14. Dezember 2009

Einleitung

Prioritätsumkehr (engl. *Priority Inversion*)

► Prioritätsumkehr

- Betriebsmittel (BM) wird von nieder-priorem Job gehalten
- ~ **unvermeidbar** bei blockierender Synchronisation
 - aber **kontrollierbar**

► **unkontrollierte** Prioritätsumkehr

- BM-Freigabe wird von unbeteiligtem Job verzögert
- vgl. Mars Pathfinder

Einleitung

Synchronisation in ereignisgesteuerten Systemen

► **explizite Synchronisation** notwendig

~ daraus ergeben sich (neue) Probleme

- (unkontrollierte) **Prioritätsumkehr**
- **Verklemmungen** (Deadlocks)

~ **Lösung**: spezielle Synchronisationsprotokolle

- blockierend und nicht-blockierend
- Verdrängungssteuerung
- Prioritätsvererbung
- Prioritätsobergrenzen

Einleitung

Verklemmungen (engl. *Deadlocks*)

“Zwei oder mehr nebenläufige Aktivitätsträger warten auf einen der jeweils anderen Aktivitätsträger.”

► Wann kann es zu **Verklemmung** kommen?

► 3 **notwendige** Bedingungen

- exklusive Belegung von BM (engl. *mutual exclusion*)
- Nachforderung von BM (engl. *hold and wait*)
- kein Entzug von BM (engl. *no preemption*)

► 1 **hinreichende** Bedingung

- zirkulares Warten (engl. *circular wait*)

Non-preemptive Critical Sections (NPCS)

- ▶ keine Verdrängung innerhalb kritischer Abschnitte
- ~ verhindert **unkontrollierte** Prioritätsumkehr
- ~ verhindert **Verklemmungen**

Vorteile

- ▶ sehr simple Implementierung
- ▶ kein a-priori Wissen

Nachteile

- ▶ Blockade höher-priorer Jobs
- ▶ auch ohne BM-Konflikt

Prioritätsvererbung (engl. *Priority Inheritance* - PIP)

- ▶ Verdrängung innerhalb des kritischen Abschnitts erlaubt
- ▶ unterbrechender Job (höher-prior) blockiert an BM-Anforderung
- ▶ verdrängter Job **erbt** dessen Priorität
- ~ verhindert **unkontrollierte** Prioritätsumkehr

Vorteile

- ▶ keine unnötige Blockade
- ▶ kein a-priori Wissen

Nachteile

- ▶ komplexere Implementierung
- ▶ transitive Blockierung

Prioritätsobergrenzen (engl. *Priority Ceiling* - PCP)

- ▶ Prioritätsvererbung
- ▶ **Ordnung** über alle BM
- ~ verhindert **unkontrollierte** Prioritätsumkehr
- ~ verhindert **Verklemmungen**
- ▶ Ordnung der BM
 - ▶ **Prioritätsobergrenze** des BM $\leadsto \max(p_i)$ aller belegenden Tasks
 - ▶ **globale Prioritätsobergrenze** Π
- ▶ Belegung eines BM
 - ▶ globale Prioritätsobergrenze wird im System verwaltet
 - ▶ ändert sich mit der BM-Belegung/Freigabe
 - ▶ **Anhebung** bei der **Belegung**: $\Pi = p_{bm}$
 - ▶ **Absenkung** bei der **Freigabe**: $\Pi = \Pi_{orig}$
 - ▶ **Bedingung für das Belegen** eines BM durch den Task T_i
 - ▶ BM ist noch nicht belegt
 - ▶ Priorität $p_i > \Pi$ **oder**
 - ▶ Priorität $p_i = \Pi$ und T_i hat ein entsprechendes BM belegt

Prioritätsobergrenzen (Priority Ceiling) - Fortsetzung

Vorteile

- ▶ geringere Blockadezeit
 - ▶ keine transitive Blockade
 - ▶ max. 1 krit. Abschnitt

Nachteile

- ▶ komplexe Implementierung
- ▶ Avoidance Blocking
- ▶ a-priori Wissen

Stack-based Priority Ceiling Protocol

- ▶ Abwandlung des normalen Priority Ceiling
- ▶ Anwendung z.B. im OSEK Betriebssystem
- ~ verhindert **unkontrollierte** Prioritätsumkehr
- ~ verhindert **Verklemmungen**
- ~ Aufgabe 4

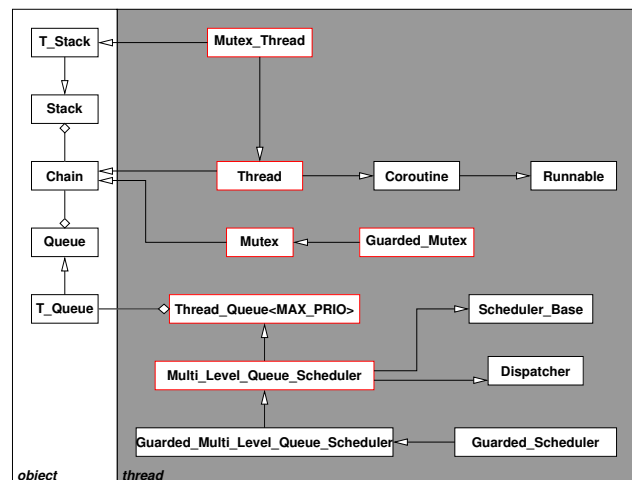
Vorteile

- ▶ einfacherer Implementierung
- ▶ effizient (kein zusätzlicher Kontextwechsel)
- ▶ kurze Blockadezeiten

Aufgabenstellung

- ▶ Implementierung des OSEK Priority Ceiling Protocol
 - ▶ Variante des Stack-based Priority Ceiling Protocol
- ▶ BM $\leadsto$ Abbildung auf Mutex
 - ▶ jede Mutex hat Prioritätsobergrenze p_{mutex}
 - ▶ Priorität des höchst-prioren Tasks, der dieses BM belegen möchte
- ▶ Operation lock(mutex)
 - ▶ Prioritätsanhebung $\leadsto p_i = p_{mutex}$
 - ▶ Bedingung: $p_i \leq p_{mutex}$ (impliziert Ordnung auf den BM)
 - ▶ Bedingung: im kritischen Abschnitt
 - ▶ kein `exit()`, `kill()` oder `yield()`
 - ▶ Bedingung: Anforderung der BM mit steigenden p_{mutex}
- ▶ Operation unlock(mutex)
 - ▶ Wiederherstellung der Priorität $\leadsto p_i = p_{i, before}$
 - ▶ Bedingung: Freigabe in umgekehrter Reihenfolge
 - ~ perfekte Schachtelung der kritischen Abschnitte

Klassenhierarchie



- ▶ **rot eingerahmte Klassen** $\leadsto$ Implementierung bzw. Ergänzung

Klassen Mutex_Thread und MLQ_Scheduler

- ▶ Klasse **Mutex_Thread**
 - ▶ ist ein Thread
 - ▶ nur ein Mutex_Thread kann BM belegen
 - ▶ Verwalten belegter BM
 - ▶ Mutex-Stack (Reihenfolge!)
 - ▶ **Achtung:** `getPriority()` ist virtuell
- ▶ Klasse **Multi_Level_Queue_Scheduler**
 - ▶ Erweiterung um `inherit_priority()`, `restore_priority()`
 - ▶ etwas wie `push_back()` wird notwendig
 - ▶ Einfügen am Anfang der Liste (vgl. bisherige Queues)
 - ▶ `restore_priority()`: Scheduler aktivieren?
 - ~ `set_need_reschedule()`

Klasse `Mutex`

- ▶ Implementiert das **OSEK Priority Ceiling Protocol**
- ▶ Faden registrieren: ❶ `...::enlist()`
 - ▶ Faden will Mutex belegen $\leadsto$ **a-priori Wissen**
 - ▶ Bestimmung der Prioritätsobergrenze
 - $\leadsto$ Aufruf in `user_main()`
- ▶ Mutex belegen: ❷❸ `...::lock()`
 - ▶ Priorität des Fadens prüfen $p_i \leq p_{mutex}$
 - $\leadsto$ evtl. Rückgabewert `false`: Verstoß gegen das Protokoll
 - ▶ Priorität vererben
 - ▶ Mutex auf dem Mutex-Stack des Fadens ablegen
- ▶ Mutex freigeben: ❹❺ `...::unlock()`
 - ▶ Mutex-Stack des Fadens prüfen
 - $\leadsto$ evtl. Rückgabewert `false`: Verstoß gegen das Protokoll
 - ▶ Mutex vom Stack nehmen
 - ▶ ursprüngliche Priorität wiederherstellen
 - $\leadsto$ `...::restore_priority()`