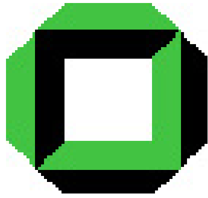


UNIVERSITÄT KARLSRUHE (TH)
Fakultät für Informatik
System Architecture Group
Frank Bellosa, Gerd Liefländer, Philipp Kupferschmied
Dominik Bruhn, Atanas Dimitrov,
Jonathan Dimond, Johannes Weiß

Basispraktikum Systemarchitektur - WS 2008/2009

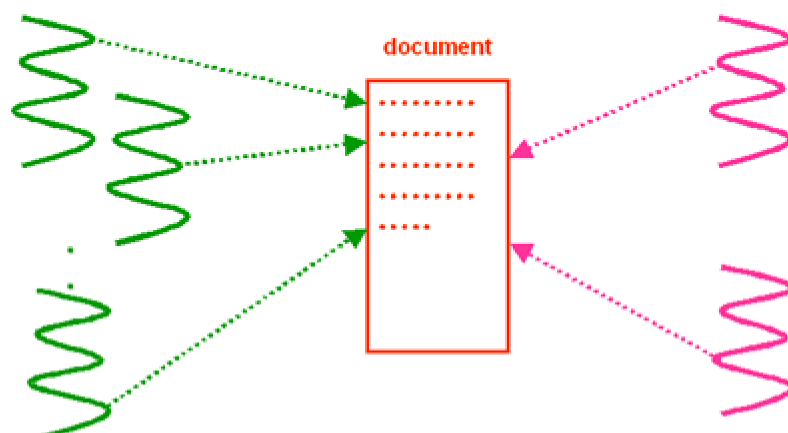
Leser/Schreiberproblem

1 Thematik

Zwecks Zusammenarbeit nebenläufiger bzw. paralleler Threads kann man sich auf so genannte gemeinsame Daten abstützen. Man kann in diesem Fall von Kooperation sprechen, da eine vernünftige Zusammenarbeit auf gemeinsamen Daten kritisch sein kann. Während das reine Lesen gemeinsamer Information unkritisch ist, ist das Verändern von gemeinsamer Information im allgemeinen kritisch und muss demzufolge durch besondere Maßnahmen koordiniert werden. Programmabschnitte, die Zugriffe auf gemeinsame Daten beinhalten, nennt man deswegen auch kritische Abschnitte (*critical sections*). Auch gleichzeitiges Lesen und Schreiben führt u.U. zu inkonsistenter Information.

Frage 1: Geben Sie ein einfaches Beispiel an, wie gleichzeitiges Lesen und Schreiben auf gemeinsamen zu inkonsistenten Daten führen kann.

Hinweis: Machen Sie sich dabei klar, wie Anweisungen einer höheren Programmiersprache in die jeweilige Maschinensprache (bei Java in den Bytecode) übersetzt werden.



Während gleichzeitig im Prinzip unendlich viele Leser auf das gemeinsame Dokument zugreifen können, soll es aus Konsistenzgründen jeweils nur genau einem Schreiber erlaubt sein, auf das Dokument zuzugreifen. Man spricht vom Lesevorrang, wenn man den oder die Schreiber solange zurückhält, bis aktuell kein Leser mehr zugreifen will, bzw. vom Schreibervorrang,

wenn nach einem Schreibwunsch kein weiterer Leser mehr zugelassen wird, bis der bzw. die Schreiber ihre Aktualisierungen des Dokuments durchgeführt haben.

Frage: Überlegen Sie sich anhand einiger konkreter Beispiele, wann Sie Leser- bzw. Schreibervorrang realisieren würden.

2 Grundlagen

Sowohl die Leser als auch die Schreiber sind als Java-Threads zu implementieren, daher werden auch in diesem Versuch die Java-Synchronisationsdirektive zwingend benötigt, um die Aufgabe korrekt zu lösen.

2.1 Java Monitore

Ein historisch relativ früh entwickeltes Sprachkonzept zur Zwangsserialisierung von kritischen Abschnitten sind Hoare's-Monitore. Ein Monitor entspricht einem Objekt, dessen Schnittstellenfunktionen (*member functions*) unter gegenseitigem Ausschluss stehen, d.h. so beschaffen sind, dass höchstens ein Thread eine Monitorschnittstellenfunktion ausführen kann. Mittels des Konzepts der synchronisierten Klassen (*synchronized*) kann man in Java ein Monitorobjekt modellieren. Hierzu sollten sie Monitorprozeduren wie z.B. `read()`und `write()` konstruieren, die das Lesen, bzw. Schreiben auf dem gemeinsamen Dokument modellieren.

Der Fall, dass z.B. gelesen werden soll währen gerade ein Schreiber am Werk ist muss abgefangen werden, d.h. Sie müssen innerhalb der entsprechenden Prozedur dies überprüfen und gegebenenfalls den entsprechenden Thread blockieren. Ein blockierter Thread konkurriert solange nicht mehr um den Prozessor, bis er wieder deblockiert, also bereit (*runnable*) wird. Ein Thread kann mittels der Funktion *wait()* an einer Bedingungsvariablen (*condition variable*) blockiert werden und so das Monitorobjekt verlassen, bis durch ein *notify()* bzw. *notifyAll()* in einem anderen Thread dieser blockierte Thread wieder deblockiert wird. Wie üblich garantiert Java nicht, dass der am längsten angehaltene Thread durch ein *notify()* bzw. *notifyAll()* aufgeweckt wird, es wird irgendein wartender Thread bei *notify()* deblockiert.

Damit dieser Fall überhaupt auftreten kann, dürfen Sie die Leser und Schreiber nicht auf das selbe Objekt synchronisieren (lesen Sie hierzu die Java API) – denn sonst ist immer nur ein Leser oder Schreiber aktiv

Eine typische Programmstelle innerhalb einer synchronisierten Monitorprozedur eines Monitorobjekts sieht wie folgt aus:

while (condition) try {wait();} catch (Interrupted Exception e} {}

Man beachte, dass an obiger Programmstelle der aufrufende Thread angehalten wird, was gleichbedeutend mit einer Threadumschaltung ist.

Hinweis: *Falls Sie Probleme mit der Java-Syntax haben, dann lesen Sie das Java-Tutorial zum Thema Threads (The Java Tutorial / Essential Java Classes / Threads: Doing Two or more Tasks at Once). Die URL lautet zur Zeit:* http://java.sun.com/docs/books/tutorial/essential/threads/

3 Experimente

3.1 First Man Standing

Implementieren Sie die benötigte Umgebung, um den Versuch zu realisieren, also Schreiber- und Leser Threads sowie das gemeinsam genutzte „Dokument“ und eine aussagekräftige Visualisierung. Testen sie ihre Implementierung zunächst ruhig nur mit einem Leser und einem Schreiber.

3.2 „3.1++“

Versichern Sie sich, dass Sie die Synchronisationsaufgabe richtig gelöst haben, indem Sie nun beliebig viele Leser und beliebig viele Schreiber zulassen und ihre Simulation auf korrekte Funktionsweise prüfen. Zudem soll man zwischen den verwendeten Zugriffsprotokollen „Leservorrang“ und „Schreibervorrang“ wechseln und auch die Anzahl der Leser- und Schreiberthreads einstellen können; wobei die Einstellungen nicht zwingend während der Simulation änderbar sein müssen; eine Konfiguration zu Beginn der Simulation reicht zur Not aus.

Hinweis: 5 < „beliebig“< 25.