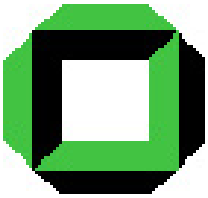

UNIVERSITÄT KARLSRUHE (TH)
Fakultät für Informatik
System Architecture Group
Frank Bellosa, Gerd Liefländer, Philipp Kupferschmied
Dominik Bruhn, Atanas Dimitrov,
Jonathan Dimond, Johannes Weiß

Basispraktikum Systemarchitektur - WS 2008/2009

Erzeuger-/Verbraucherproblem

1 Thematik

In der Literatur zum parallelen Programmieren (*concurrent programming*) ist das so genannte Erzeuger/Verbraucherproblem ein Standardbeispiel für Interaktionen von parallelen Prozessen. Stellen Sie sich ein Lager begrenzter Kapazität vor, in das ein Nahrungsmittelproduzent eines fiktiven Produkts seine Waren ablegt, aus dem sich ein Verbraucher bei Bedarf bedienen kann. Saisonal bedingt kann es auf beiden Seiten zu Engpässen kommen, so dass die Fälle abgefangen werden müssen, dass ein Produzent Waren in ein volles Lager ablegen bzw. ein Konsument sich aus einem leeren Lager bedienen möchte. Mögliche Spielarten dieses Problems sollen in den folgenden Experimenten mit den von Java angebotenen Mechanismen gelöst werden.

2 Grundlagen

Entwerfen Sie eine Java-Applikation, die aus dem Hauptprogramm heraus ein Lager begrenzter Kapazität als globales Ganzzahlfeld *buffer* anlegt (siehe auch Versuch 1). Anschließend sollen e≥1 Erzeuger und v≥1 Verbraucher als zyklische Threads erzeugt werden. Die Größen e und v, sowie die Lagerkapazität sollen **Eingabeparameter** sein. Jeder Erzeuger durchläuft genau n≥1 mal einen Erzeugungszyklus. Verbraucher terminieren, wenn kein Erzeuger mehr lebt und das Lager leer ist. Erzeuger inkrementieren einen mit 1 initialisierten **globalen** Zähler, legen dessen Wert in *buffer* ab, um sich dann kurzfristig auszuruhen (mittels *sleep(mit interaktiv veränderbarer sleepdauer)*). Verbraucher entnehmen aus diesem *buffer* den Wert, schreiben an dessen Stelle wieder den Wert 0 und addieren ihn auf, um sich ebenfalls für eine Weile auszuruhen. Das Lager *buffer* habe einstellbar viele Ablageplätze und werde als Ringpuffer benutzt. Der i-te Verbraucher legt seine jeweilige Zwischensumme im globalen Feld *result[i]* ab.

2.1 Modellüberlegungen

Wie man aus den Experimenten des ersten Versuchs erkennen musste, kann man bei einer Problemlösung mittels nebenläufiger oder paralleler Threads im allg. nicht voraussagen, in welcher Reihenfolge diese Threads abgearbeitet werden. Sobald man aber für eine korrekte Problemlösung eine ganz bestimmte Abfolge der Threads benötigt, bedarf es zusätzlicher **koordinierender** Maßnahmen, die die eher zufällige Abarbeitungsfolge der Threads in ein deterministisches Interaktionsmuster umwandeln.

Frage 1: Wann spricht man von nebenläufigen, wann von parallelen Threads?

Zusätzliche Probleme können bei nebenläufigen bzw. parallelen Threads auftreten, wenn gemeinsame Daten oder gemeinsam benutzte Betriebsmittel ins Spiel kommen. Man stelle sich folgende Situation vor:

Ein Thread verändere einen Datensatz, der aus mehreren Einträgen bestehen möge. Ein anderer Thread soll für die Ausgabe dieses Datensatzes verantwortlich sein. Wenn beide Threads ihre Arbeit unkoordiniert verrichten, kann es passieren, dass der Ausgabethread einen noch nicht vollständig aufbereiteten Datensatz ausgibt. Man spricht im allg. in solchen Fällen von **Wettbewerbssituationen** (*race conditions*).

Frage 2: Welche Wettbewerbssituationen kann es im allgemeinen Fall im obigen Erzeuger-/Verbraucherproblem geben?

Frage 3: Wenn e und v jeweils 1 sind, kann man dann bei korrekter Implementierung voraussagen, was in result[1] bei Terminierung von Verbraucher i steht?

2.2 Java-Konzepte zur Koordinierung von Threads

Java bietet verschiedene Sprachkonzepte zur Koordinierung von Threads an, vergleichen Sie hierzu die Methoden der Klasse *thread*.

Frage 4: Welche Sprachkonzepte sind damit gemeint?

Frage 5: Wie könnte man unabhängig von e und v eine strikt abwechselnde Abarbeitung von Erzeugern und Verbrauchern erzwingen?

Frage 6: Wovon hängt die Lagerkapazität ab, d.h. wann braucht man ein größeres Lager, wann kommt man mit einen kleineren aus?

3 Experimente

3.1 Einfaches Erzeuger-/Verbraucherproblem mit nur einem Puffer

Man löse dieses einfache Erzeuger-/Verbraucherproblem so mit den entsprechenden Java-Konzepten, dass sich Erzeuger und Verbraucher **strikt** abwechseln, wobei sie jeweils genau eine Erzeuger- bzw. Verbraucherphase durchlaufen. Die Abarbeitung der beiden Threads sowie des gemeinsamen einelementigen Lagers visualisiere man durch geeignete Textausgaben.

3.2 Einfaches Erzeuger-/Verbraucherproblem mit b Puffern

Unter Ausnützung des Datenlagers **buffer[0,b-1]** löse man das einfache Erzeuger-/Verbraucherproblem so, dass mal mehrere Erzeuger- mal mehrere Verbraucherphasen hintereinander bearbeitet werden können. Hierzu platziere man in den beiden Threads an geeigneter Stelle Aufrufe der Funktion *sleep()*. Die Länge der Schlafphasen der beiden Threads gestalte man so, dass sie per Kommandozeilenparameter eingestellt werden können. Diese Schlafphasen sollen das eigentliche Erzeugen bzw. Verbrauchen simulieren, sollen also irgendwie erkennbar sein. Muss hingegen ein Erzeuger warten, weil kein Pufferplatz mehr zur Verfügung steht, dann soll auch dies erkennbar sein; ebenso beim Verbraucher. Lassen Sie sich zudem etwas einfallen, wie man schnell erkennen kann, dass kein Produkt verloren wurde.

3.3 Allgemeines Erzeuger-/Verbraucherproblem mit b Puffern

Analog zu Experiment 3.2 soll nun das allgemeine Erzeuger-/Verbraucherproblem gelöst werden, wobei gelten soll: **e+v ≤ 10**. Zusätzlich soll nun die ganze Situation lebendig visualisiert werden. Lassen Sie sich hierfür zunächst eine gut überschaubare Darstellung für die Erzeuger, die Verbraucher und das Lager einfallen. Außerdem sollen nun alle Parameter wie die Schlafzeit von Erzeugern und Verbrauchern zur Laufzeit änderbar sein.

Wichtig: Die ganze Situation muss vollständig ohne Textausgabe auf der Konsole erfassbar sein sowie alle relevanten Daten anzeigen.