Einleitung

In diesem Versuch handelte es sich um das Producer/Consumer-Problem, wobei nach einer konsistenten Lösung mit durchschaubaren Resultaten gefragt war. Das Producer/Consumer-Problem war in Hinsicht mehrer Szenarien zu lösen, die im Folgenden näher erläutert werden.

Source

Hier soll die Implementierung unserer Lösung beschrieben werden, es wird ausführlich auf die Methoden, die Klassen, derer Funktion und deren Beziehungen eingegangen.

Launcher ist die Klasse, die den Auftrag hat die Benutzereingabe zu überprüfen, zu verarbeiten, ggf. den Benutzer zu orientieren und anschliessend den gewünschten Versuch erfolgreich zu starten. Sie instanziiert Klassen wie *Buffer*, *Producer*, *Consumer* und *Gui*.

Ein *Producer* ist ein Thread, der periodisch einen mit 1 initialisierten globalen Wert, den sogennanten Zähler (globalCounter, lokalisiert in der *Buffer* Klasse), in der nächsten verfügbaren Position eines globalen Ringpuffers (*CircularBuffer*, auch befindend in der Klasse Buffer) setzt und ihn anschliessend um 1 erhöht. Dazu ruft er die Methode put() der Klasse *Buffer* auf, indem er den zu-schreibenden Wert und seine Kennzahl (Thread ID) der Methode übergibt. Wie diese Methode arbeitet, sei dem Abschnitt über die Klasse *Buffer* zu verweisen. Nachdem der Producer geschrieben hat, wird er für eine bestimmte Zeit im schlafenden Zustand gelegt. Während des Schreibens wird der Thread als occupied (beschäftigt) vermerkt, nach dem Schlafen wird er als nicht-beschäftigt zurückgesetzt. Warum das notwendig ist, sei dem *Gui* Abschnitt zu verweisen. Der Producer wiederholt diese Schritte bis er die Anzahl der ihm erlaubten Laufzyklen (cycles) erreicht hat. Diese Klasse wird als Producerklasse in allen 3 Experimenten dieses Versuchs verwendet.

Ein *Consumer* ist ein Thread, der, ebenfalls periodisch, den globalen Zähler der Klasse *Buffer* ausgibt. Er liest den aktuellen Wert aus dem Ringpuffer aus (dazu nutzt er die Methode get() der *Buffer* Klasse, der er seine Kennzahl, sog. Thread ID, übergibt) und legt sich für eine gewisse Zeit schlafen. Wie bei den *Producer*s wird er während des Lesens als occupied und nach dem Schlafen als nicht-occupied vermerkt. Der *Consumer* läuft bis kein *Producer* mehr lebt und bis der Ringpuffer völlig entleert wurde.

Jedem *Producer*- und *Consumer*objekt ist ein Objekt des Typs *ThreadControlBlock* zugeordnet, die aktuellste Informationen über den Zustand und Eigenschaften des dazugehörigen Threads enthält. Über diese Klasse kann man dem ihr entsprechenden Thread (Producer thread bzw. Consumer thread1), der Zykluszeit (int cycleLength) und Anzahl der Zyklen des Threads (int cycles) und dem Zustand ob er fertig (boolean done), wartend oder aktiv ist (boolean occupied), ansprechen bzw. diese manipulieren.

Die Klasse *Buffer* ist die zentrale Klasse, die ausschlaggebend für die ganze Threadaktivität ist, wo die *Producer*s und *Consumer*s „Produkte" ablegen bzw. abholen, wo die Entscheidung getroffen wird ob es produziert oder verbraucht wird und die Klasse, die die ganzen notwendingen Synchronisationsmechanismen für diesen Versuch enthält. Eine Beschreibung der wichtigsten Attributen und Methoden folgt: ArrayList<Integer> results ist das Vektor, dass den Arbeitsgehalt der *Consumer* speichert. Der Arbeitsgehalt wird dadurch berechnet, dass jedes Mal, wenn ein *Consumer* ein Produkt (globalCounter) aus dem *Buffer* abholt, sein Wert zu einer mit 0 initialisierten Variable result hinzuaddiert

wird. Der Arbeitsgehalt eines *Consumer*s, der die Thread ID i hat, befindet sich im i-ten Slot des Vektors. ArrayList<Producer> producerList bzw. ArrayList<Consumer> speichert die *Producer*s bzw. die *Consumer*s. Int producersCounter wird mit 0 initialisiert und zählt die *Producer*s auf, die nicht beendet sind. Die Methode stopAll() beendet alle *Producer*/*Consumer* Threads. Die beiden registerThread()-Methoden fügen die *Consumer*s/*Producer*s den entsprechenden Listen hinzu, erzeugen die jeweiligen *ThreadControlBlock*s und anschliessend starten die Threads. Wir haben uns in diesem Versuch dafür entschieden, die wait()/notifyAll() Synchronisationsmechanismen der Sprache Java zu verwenden, die in den put() und get() Methoden eingebaut sind. Die put () Methode dient dazu, einen Wert in dem *CircularBuffer* zu schreiben. Die Methode kontrolliert zunächst, ob noch Platz frei zum Schreiben in dem *CircularBuffer* ist. Wenn ja, dann setzt sie fort, schreibt den Wert, gibt eine Bestätigung aus und anschliessend teilt den anderen Threads mit, dass sie laufen dürfen, ansonsten wartet die Methode bis Plätze freigemacht werden. Die Methode get(), die den Zweck hat aus dem *CircularBuffer* „Produkte" zu holen, folgt dem gleichen Prinzip: falls sich keine „Produkte" im *CircularBuffer* befinden, wartet sie, bis es welche gibt, ansonsten holt sie ein „Produkt" ab, aktualisiert den Arbeitsgehalt des Threads, der sie aufgerufen hat, gibt eine Bestätigung aus und teilt den anderen Threads, dass sie laufen dürfen. notifyAll() weckt alle Threads auf, sowohl *Consumer* als auch *Producer* konkurrieren miteinander um das lock des Objekts vom Typ *Buffer* zu kriegen, wer das kriegt, das entscheidet letztendlich der VM-Scheduler und ob er put()- und get()-Operationen ausführen darf, wird in den letzteren beschlossen. Somit ist nicht ausgeschlossen, dass ein Thread bei mehrstelligem CircularBuffer oder dass mehrere *Consumer* und *Producer* mehrere Zyklen hintereinander ausführen.

Die Klasse *CircularBuffer* ist die eigentliche Datenstruktur, wo der globale Zähler, also die „Produkte", (int globalCounter) gespeichert wird und die, die Schreibe- und Leseoperationen steuert. Zur Speicherung der Daten wird ein Array fester Größe des primitiven Typs int (int[] buffer) verwendet. Die Zeiger writePointer bzw. readPointer indizieren die nächste Stelle des Arrays, in/aus der geschrieben bzw. gelesen werden kann. Nach einer Schriebe- bzw. Leseoperation werden die jeweiligen zeiger um 1 inkrementiert. Die grundlegenden Methoden sind writeBuffer() und readBuffer(). Nach dem Lesen des Arrays, wird die Stelle, aus der gelesen wurde, auf 0 gesetzt (wird also als „leer" markiert). Der *CircularBuffer* ist leer, wenn die Schreibe- und Lesezeiger gleich sind, d.h. es wurde soviel geschrieben wie es gelesen (und entleert) wurde. Dazu dient die Methode isEmpty(). Die Methode isFull() überprüft, ob der Array vollbesetzt ist. Sie liefert genau dann „wahr" zurück, wenn die Differenz zwischen dem writePointer und readPointer gleich die Kapazität des Arrays ist. getNumberOfFullSlots() liefert einfach die Anzahl der besetzten Stellen des Arrays.

Die Klasse *Gui* wird zur Visualisierung im 3. Experiment verwendet. Hilfsklassen sind *ProducerRep*, *ConsumerRep*, *StorageRep* und *Updater*. Diese Klasse stellt eine Schnittstelle bereit, wodurch Experimentparameter graphisch eingegeben werden können. Alle Parameter bis auf die Lagerkapazität sind im Laufe des Experiments dynamisch veränderbar. Die Klasse *Gui* visualisiert die Komponenten *ProducerRep*, *ConsumerRep* und *StorageRep*. *ProducerRep* ist eine graphische Räpresentierung der *Producer*. Für jeden *Producer* wird ein Rechteck angelegt, der mit dem aktuellen Zustand des Threads beschriftet ist. Jeder Zustand („producing", „waiting" und „done") wird durch eine Farbe vertreten (jeweils grün, rot und grau). Der Zustand des Threads wird der Klasse ThreadControlBlock (den Attributen boolean occupied, boolean done) entnommen. Analog ist auch die Klasse *ConsumerRep* aufgebaut. *StorageRep* ist dafür ausgelegt, den Bufferinhalt graphisch zu interpretieren. Die Klasse *Updater* aktualisiert die Komponenten *ProducerRep*, *ConsumerRep* und *StorageRep* in regelmäßigen Zeitabständen.

Experimente

3.1 Einfaches Erzeuger-/Verbraucherproblem mit nur einem Puffer

Implementierung:

S. *Launcher*, *Buffer*, *Producer*, *Consumer*, *CircularBuffer*, *ThreadControlBlock*.

Kurze Experimentsbeschreibung:

Es sollte das einfache Erzeuger-/Verbraucher Problem für 1 Producer und 1 Consumer mit einem Pufferplatz gelöst werden. Ergebnisse waren in der Konsole anzuzeigen.

Ergebnisse:

Die Producers und Consumers laufen strikt abwechselnd ab. Es kann nie passieren, dass mehrere Producer- bzw. Consumerphasen hintereinander durchgeführt werden, da es nur einen Platz im Puffer gibt. Die Ablauffolge der Threads und das Endergebnis sind deterministisch. Der Arbeitsgehalt des Consumer-Threads ist nicht zufallsbedingt und kann ausgehend von der Anzahl der Zyklen vorausgesagt werden. Ausserdem ist in der Ausgabe erkennbar, dass keine Information verloren geht.

3.2 Einfaches Erzeuger-/Verbraucherproblem mit b Puffern

Implementierung:

S. *Launcher*, *Buffer*, *Producer*, *Consumer*, *CircularBuffer*, *ThreadControlBlock*.

Kurze Experimentsbeschreibung:

Es sollte das Einfache Erzeuger-/Verbraucher Problem für 1 Producer und 1 Consumer mit mehreren Pufferplätze gelöst werden. Ergebnisse waren in der Konsole anzuzeigen.

Ergebnisse:

Die Producers und Consumers laufen nicht mehr strikt abwechselnd ab. Gibt man eine größere Zykluszeit für den Consumer als Parameter ein, so werden mehrere Producephasen hintereinander ausgeführt bis der Buffer vollständig besetzt ist. Danach laufen der Consumer und Producer abwechselnd ab, da der Producer warten muss bis der Consumer ein Produkt aus dem Lager herausgeholt hat. Gibt man die gleiche Zykluszeit für den Consumer und den Producer ein, dann laufen diese fast abwechselnd, da nachdem der Producer produziert hat, sich schlafen legen und warten muss, der Consumer kommt aber bevor er sich aufwacht und holt den Wert vom Buffer ab. Der Consumer schläft dann selbst auch, mittlerweile ist der Producer wach und legt wieder ein Produkt in den Buffer ab, usw. Experiment lief wie erwartet, wiederum ist der Arbeitsgehalt des Consumers voraussagbar, da es nur einen Consumer gibt, die eine Arbeit von feststehendem Aufwand, erledigen muss. In der Konsole erkennt man, dass die Implementierung konsistent und dadurch geht keine Information unterwegs verloren.

3.3 Allgemeines Erzeuger-/Verbraucherproblem mit b Puffern

Implementierung:

S. *Launcher*, *Buffer*, *Producer*, *Consumer*, *CircularBuffer*, *ThreadControlBlock*, *Gui, ProducerRep, ConsumerRep, StorageRep, Updater*.

Kurze Experimentsbeschreibung:

Es sollte das allgemeine Erzeuger-/Verbraucher Problem für e Producer und v Consumer (e+v <= 10) mit mehreren Pufferplätze gelöst werden. Alle relevanten Ergebnisse und Informationen waren diesmal graphisch anzuzeigen. Konsolausgabe dürfte nicht verwendet werden.

Ergebnisse:

Ähnlich wie im vorigen Experiment, laufen die Producers schneller, so wird der Puffer irgendwann voll und die Producers und Consumers müssen abwechselnd laufen. Die Producers müssen auf die Consumers warten, bis diese Produkte rausholen. Laufen dagegen Consumers schneller, so müssen die Consumers auf die Producers warten. Producers und Consumers laufen also wieder abwechselnd. Ein weiteres Faktor, das die Geschwindigkeit mit der Pufferplätze besetzt werden beeinflusst, ist die Anzahl der laufenden Consumers bzw. Producers. Auch hier ist Konsistenz gewährleistet.

Fragen beantworten

Modellüberlegungen

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

Antwort 1: Bei nebenläufigen Threads geht es um Threads, die in keinerlei Relation miteinander stehen bzw. sich nicht irgendwie beeinflussen oder nicht voneinander abhängen (z.B. Threads, die nicht miteinander kooperieren). Parallele Threads sind das Gegenteil davon.

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

Antwort 2: Ein Consumer kann einen noch nicht veränderten Datensatz auslesen, merkt aber, dass er nicht Produkte holen darf (z.B. weil es da keine Produkte noch gibt) und fängt an, zu warten. Bevor der Consumer wartet, kann ein Producer anfangen, Produkte zu produzieren und die in den Datensatz schicken. Ist er fertig, teilt er dem Consumer mit, dass er laufen soll (z.B. mittels notifyAll()). Da er noch nicht wartend ist, geht diese Mitteilung verloren, der Consumer fängt an zu warten, und wird nie aktiv.

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

Antwort 3: Ja. Der Producer schreibt einmal, dann wartet, der Consumer kommt dran, er verbraucht, ruft den Producer auf und dann wartet selbst usw. Wenn wir wissen, wie

lange der Producer läuft (wie viele Zyklen), dann wissen wir auch was in result[i] bei Terminierung vom Verbraucher steht.

Java-Konzepte zur Koordinierung von Threads

Frage 4: Welche Sprachkonzepte sind damit gemeint?

Antwort 4: Damit sind gemeint, die wait()/notify()- bzw. wait()/notifyAll()- und die suspend()/resume()-Methoden. Ausserdem kann man in Java auch Semaphoren implementieren.

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

Antwort 5: Indem man nach einer Produktionsphase die Producers warten lässt und die Consumers aufruft und nach einer Verbrauchsphase die Consumerrs warten lässt und die Producers aufruft.

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

Antwort 6: Die Lagerkapazität hängt von der Zyklenzeit der Verbraucher bzw. Producer ab.