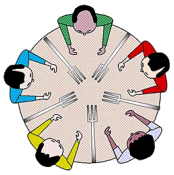
Einführung

In der theoretischen Informatik ist das Philosophenproblem ein klassisches Beispiel für die Notwendigkeit einer Prozeß-Synchronisation, sobald mehrere Prozesse bzw. Threads innerhalb einer Anwendung aktiv sind.

Es sitzen fünf Philosophen an einem runden Tisch, und jeder hat einen Teller mit Spaghetti vor sich. Zum Essen von Spaghetti benötigt jeder Philosoph zwei Gabeln. Allerdings waren im Restaurant nur fünf Gabeln vorhanden, die nun zwischen den Tellern liegen. Die Philosophen können also nicht gleichzeitig speisen.



Bevor die Philosophen speisen, denken sie in ihrem Arbeitsraum über philosophische Probleme nach. Wenn ein Philosoph hungrig wird, greift er zuerst die Gabel links von seinem Teller, dann die auf der rechten Seite und beginnt zu essen. Wenn er satt ist, legt er die Gabeln wieder zurück und beginnt wieder zu denken. Sollte eine Gabel nicht an ihrem Platz liegen, wenn der Philosoph sie aufnehmen möchte, so wartet er, bis die Gabel wieder verfügbar ist.

Solange nur einzelne Philosophen hungrig sind, funktioniert dieses Verfahren wunderbar. Es kann aber passieren, dass sich alle fünf Philosophen gleichzeitig entschließen zu essen. Sie setzen sich also auf ihten Platz und ergreifen alle gleichzeitig ihre linke Gabel und nehmen damit dem jeweils links von ihnen sitzenden Kollegen seine rechte Gabel weg. Nun warten alle fünf darauf, dass die rechte Gabel wieder auftaucht. Dies passiert aber nicht, da keiner der fünf seine linke Gabel zurücklegt. Die Philosophen verhungern. Nimmt man weiterhin an, die Philosophen seien unabhängig voneinander agierende Threads, so spricht man an dieser Stelle von einem sogenannten Deadlock .

Die klassische Aufgabenstellung soll in diesem Artikel erweitert werden. Ziel soll es sein, ein Programm in Java zu entwickeln, in dem eine beliebig große Zahl von Philosophen und Sitzplätzen definiert werden. Die Philosophen sollen individuell und damit vollkommen parallel agieren. Ein Aufseher überwacht die Nahrungsaufnahme der Philosophen und verhindert das Philosophen zu viel essen, während andere Philosophen hungern.

Synchronisationsobjekte in Java

Das Philosophenproblem wird in der Regel mit Mutexen oder Semaphoren gelöst, zum Beispiel nach dem Peterson-Algorithmus oder dem Dekker-Algorithmus. Beide Algorithmen zielen auf eine Sequentialisierung der Problematik ab.

Die Programmiersprache Java stellt intern einige elementare Eigenschaften zur Verfügung. Beginnend mit JDK 5.0 stehen prinzipiell zwei Mechanismen bereit, um einen Codeblock gegen konkurrierenden Zugriff zu schützen. Frühere Versionen von Java haben das Schlüsselwort *synchronized* für diesen Zweck verwendet; JDK 5.0 führt die Klasse *ReentrantLock* und *Semaphore* ein. Ein Großteil der relevanten Klassen für die Programmierung mit konkurrierenden Threads sind im Paket java.util.concurrent zu finden.

Konzeptionell verwaltet eine Semaphore eine Anzahl von Berechtigungen. Um nach einer Semaphore fortzufahren, fordert ein Thread durch Aufruf von *acquire* eine Berechtigung an. Es ist eine feste Anzahl von Berechtigungen verfügbar, was die Anzahl der Threads einschränkt, die passieren dürfen. Andere Threads können Berechtigungen durch Aufruf von *release* erteilen. Es gibt keine eigentlichen Berechtigungsobjekte. Die Semaphore führt lediglich einen Zähler. Darüber hinaus muss eine Berechtigung nicht durch den Thread freigegeben werden, der sie angefordert hat. In der Tat kann jeder beliebige Thread eine beliebige Anzahl von Berechtigungen erteilen. Wenn er mehr als die maximal verfügbaren Berechtigungen erteilt, wird die Semaphore einfach auf den maximalen Zählerstand gesetzt. Durch diese Verallgemeinerung sind Semaphore sowohl sehr flexibel als auch möglicherweise verwirrend.

Semaphoren wurden 1965 von Edsger Dijkstra erfunden, um sie als Synchronisierungs-Primitive zu verwenden. Dijkstra hat gezeigt, dass sich Semaphoren sehr effizient implementieren lassen und dass sie ausreichend leistungsfähig sind, um die gebräuchlichsten Probleme der Thread-Synchronisierung zu lösen. Die Nutzungsoperationen wurden von Dijkstra mit P und V bezeichnet. Dies sind Initialen niederländischer Wörter bzw. Kofferwörter für prolaag und verhoog. Programmierschnittstellen verwenden mnemonisch deutlichere Bezeichnungen wie wait, acquire oder down für die P-Operation und signal, release oder up für die V-Operation. Bei einem Aufruf der P-Operation wird der Zähler dekrementiert. Ist

der Zähler danach größer gleich 0, so setzt der Prozess seine Aktionen fort. Ist der Zähler jedoch kleiner als 0, kehrt der Kontrollfluss nicht aus der Operation zurück. Der aufrufende Prozess wird blockiert und in die Warteschlange des Semaphors eingereiht. Bei einem Aufruf der V-Operation wird der Zähler inkrementiert. Es wird ein Prozess aus der Warteschlange entnommen und entblockiert, falls die Warteschlange nicht leer ist. Der entblockierte Prozess setzt dann seine Aktionen mit denen fort, die dem P-Aufruf folgen, der den Prozess blockierte.

Das Philosophenproblem

Für das Philosophenproblem existieren mehrere Lösungsansätze, die im Folgenden näher erörtert werden sollen.

Dirigenten

Eine relativ einfache Lösung besteht darin einen Aufseher an einem Tisch zu platzieren. Die Philosophen müssen den Aufseher um Erlaubnis fragen, bevor sie Gabeln in die Hand nehmen dürfen. Da der Aufseher immer weiß, welche Gabeln gerade verwendet werden, kann er Verklemmungen vermeiden. Wenn an einem Tisch mit fünf Sitzplätzen vier Gabeln verwendet werden, muss der nächste Philosoph auf die Erlaubnis des Aufsehers warten, die erst dann erteilt wird, wenn eine weitere Gabel frei wird. Die Logik ist einfach gehalten, indem sichergestellt wird das ein Philosoph immer zuerst die linke Gabel hebt, bevor er die rechte Gabel nimmt oder vice versa.

Um die Funktionsweise zu verdeutlichen, stellen Sie sich vor die Philosophen werden im Uhrzeigersinn von A bis E benannt. Wenn A und C essen, sind vier Gabeln in Gebrauch. B sitzt zwischen A und C und besitzt keine Gabel, während D und E eine ungenutzte Gabel zwischen ihnen liegen haben. Stellen Sie sich weiterhin vor, D möchte nun essen. Hebt er die fünfte Gabel, kann es theoretisch zu Verklemmungen kommen. Stattdessen fragt er zuerst beim Aufseher nach und wird angewiesen zu warten. Damit lässt sich sicherstellen, dass sobald zwei Gabeln wieder zurück auf den Tisch gelegt werden, mindestens ein Philosoph erfolgreich sein Gabelpaar aufheben kann. Eine Verklemmung kann sich auf diese Weise nicht mehr ereignen.

Ressourcen-Hierarchie

Eine weitere Strategie bei der Implementierung einer thread-sicheren Lösung ist die Zuteilung einer festen Ordnung für die Ressourcen, in dem Fall die Gabeln. Dabei wird die Konvention umgesetzt, dass alle Ressourcen in einer bestimmten Reihenfolge angefordert und in der umgekehrten Reihenfolge wieder abgegeben werden. Zwei Ressourcen dürfen nie unabhängig von der Konvention gleichzeitig in Anspruch genommen werden. Bei den speisenden Philosophen sind die Ressourcen die Gabeln, die von 1 bis 5 durchnummeriert werden. Jeder Philosoph muss immer die Gabel mit der niedrigeren Nummer zuerst aufheben. Nachdem er

gegessen hat muss er in umgekehrter Reihenfolge die Gabel mit der höheren Nummer zuerst auf den Tisch legen. Sollten alle fünf Philosophen versuchen gleichzeitig eine Gabel aufzuheben, so wird die Gabel mit der Nummer 5 auf dem Tisch verbleiben, da es keine andere Gabel gibt die eine höhere Nummer besitzt. Der fünfte Philosoph wird deshalb keine Gabel aufheben können. Nur ein Philosoph hat Zugriff auf die Gabel mit der höchsten Nummer, so kann der Philosoph mit zwei Gabeln essen. Sobald ein Philosoph fertig ist, legt er die Gabel mit der höheren Nummer zuerst ab, danach folgt die andere Gabel. Damit kann der nächste Philosoph die Gabel für sich beanspruchen und mit dem Mahl beginnen.

Diese Lösung für das Philosophen-Problem wurde ursprünglich von Dijkstra vorgeschlagen.

Obwohl die Lösung über eine Ressourcen-Hierarchie Verklemmungen vermeidet, ist sie nicht immer praktikabel, insbesondere dann nicht, wenn die Liste der benötigten Ressourcen zu Beginn nicht vollständig bekannt ist. Zum Beispiel, wenn ein Philosoph die Gabel 3 und 5 hält und dann feststellt das er die Ressource 2 benötigt. In diesem Fall muss er zuerst die Gabel 5 ablegen, anschließend Gabel 3 und dann Gabel 2 anfordern, bevor er Gabel 3 wieder anfordern kann. Programme die Einträge mit hohen Nummerierungen in Datenbanken anfordern, können auf diese Weise nicht effizient arbeiten, da sie immer erst alle höheren Ressourcen lösen müssen, bevor sie eine niedere Ressource anfordern können.

Monitore

Unter der Verwendung von Monitoren können die Philosophen nur dann essen, wenn kein Nachbar gerade isst. Diese Lösung ist vergleichbar mit einem System in dem die Philosophen, die nicht die zweite Gabel erfolgreich aufheben können, die erste Gabel zuerst ablegen müssen, bevor sie es erneut versuchen können. In dem nachfolgenden Beispiel ist das mithilfe von Semaphoren realisiert.

[code xml:lang="java"]import java.util.concurrent.Semaphore; import java.util.Random; /** * Five Philosopher problem: How to model limited shared resources * http://en.wikipedia.org/wiki/Five_philosophers * This is an update of the original version posted on WikipediA * JEB 11-04-2010 */ public class Philosopher extends Thread { /** * Constructor: an ID# and two shared resources * @param i * @param fork1 * @param fork2 */ public Philosopher(int i, Semaphore fork1, Semaphore fork2) { id = i; myFork = fork1; myNeighborsFork = fork2; } /** * "Lazy" message queue. Original program used a Vector to * queue the events and displayed them at the end. I like having * feedback while the program is running, but the messages are * sometimes displayed out of order - no biggie. * * @param str */ private void postMsg(String str) { System.out.printf("%d %d Philosopher %d %sn", System.currentTimeMillis(), ++event, id, str); } /** * Pause - waits a bit (random fraction of a second) */ private void pause() { try {

sleep(rand.nextInt(1000)); } catch (InterruptedException e){} } /** * Tell philosopher to think - he waits a bit * */ private void think() { postMsg("is thinking"); pause(); } /** * Tell philosopher to eat. Tries to acquire resources (forks) * * Possible modification: Doesn't change a state * (hungry, starving, etc.) if they can't get a fork * * Possible modification: could return a boolean indicating success */ private void trytoeat() { postMsg("is hungry and is trying to pick up his forks"); pause(); try { // Semaphore - waits on his own fork if necessary myFork.acquire(); // He's picked up his own fork, now try and grab his neighbor's fork // (does not wait) if (!myNeighborsFork.tryAcquire()) { // Unsuccessful, guess he's fasting today postMsg(">>>> was not able to get his neighbor's fork"); return; }; // Success! begins to eat postMsg("picked up his forks and is eating meal #" + (10 - --meals)); pause(); // Now put down the forks postMsg("puts down his forks"); myNeighborsFork.release(); } catch (InterruptedException e) { // In case the thread is interrupted postMsg("was interrupted while waiting for his fork"); } finally { // always puts his own fork back down myFork.release(); } } /** * philosophise until all meals are consumed */ @Override public void run() { while (meals > 0) { think(); trytoeat(); } } /** * Main program * * Create resouces (forks) as semaphores * * create philosophers * * start philosophers * * wait for completion */ // Main program public static void main(String[] args) { System.out.println("Begin"); final int N = 5; // five philosophers, five forks // Create the forks, 1 fork per philosopher Semaphore[] fork = new Semaphore[N]; for (int f = 0; f In Abwesenheit von konkreten Sperralgorithmen für die Gabeln müssen die Philosophen sicherstellen das ihre Entscheidung mit dem Essen zu beginnen nicht auf einer Momentaufnahme des Zustands ihrer Nachbarn beruht. Wenn der Philosoph 1 den Philosoph 2 nicht gerade essen sieht und sich dann Philosoph 3 zuwendet, um festzustellen ob dieser gerade auch nicht isst, dann kann in der Zwischenzeit der Philosoph 2 mit dem Essen begonnen haben. Die Lösung vermeidet diesen Situation, indem sie einen wechselseitigen Ausschluss bzw. Mutex (Abk. für engl. mutual exclusion) verwendet. Dieser betrifft nicht die Gabeln, sondern die Ermittlung des Zustands der Philosophen. An diese Stelle treten Monitore. Der Test ob die Nachbarn gerade speisen, sowie das Aufnehmen und Ablegen von Gabeln obliegt dem Monitor und dem wechselseitigen Ausschluss. Philosophen, die nicht essen, halten keine Gabel in der Hand. Wenn ein Monitor einem Philosophen gestattet fortzufahren, wird dieser zunächst die erste Gabel in die Hand nehmen und anschließend die zweite Gabel. Nach dem Essen, wird der Philosoph den Monitor signalisieren das beide Gabeln nun wieder zur Verfügung stehen.

Beachten Sie das diese Lösung nicht das Verhungern von Philosphen unterbindet. Beispielsweise kann der Phliosoph mit der Nummer 2 ewig warten, wenn Philosoph 1 und 3 sich bei der Nahrungsaufnahme überschneiden.

Um ein Verhungern von Philosophen zu vermeiden, kann ein separater Zähler eingeführt

werden, der die Übersicht über die Zahl der hungernden Philosophen behält und bei ungleicher Nahrungsverteilung einschreitet.

Chandy / Misra

1984 schlugen K. Mani Chandy und J. Misra eine andere Lösung für eine beliebig große Zahl an Philsophen (benannt nach P1, ..., Pn) vor. Die Lösung eignet sich für verteilte Systeme und benötigt keine zentrale Verwaltungsstelle nach der Initialisierung.

- Für jedes Philosophenpaar, welches um eine Ressource konkurriert, generiere eine Gabel und gebe sie dem Philosophen mit der niedrigeren ID. Jede Gabel kann *sauber* oder *schmutzig* sein. Zu Beginn sind alle Gabeln schmutzig.

- Wenn ein Philosoph einen Satz von Ressourcen anfordert, muss er die Gabeln von seinen konkurrierenden Nachbarn anfordern. Für alle Gabeln, die er selbst nicht besitzt, muss er eine Anforderung senden.
- Wenn ein Philosoph mit einer Gabel eine Anforderung empfängt, behält er die Gabel, falls sie sauber ist und legt sie ab, wenn sie schmutzig ist. Wenn er die Gabel weiterreicht, dann reinigt er sie vorher.
- Nachdem ein Philosoph mit dem Essen fertig ist, werden alle seine Gabeln schmutzig. Falls ein anderer Philosoph zuvor eine Anforderung gesendet hat, reinigt der Philosoph die Gabel und reicht sie weiter.

Diese Lösung ermöglicht einen hohen Grad an Parallelität und löst das Problem für eine beliebig große Zahl an Philosophen.

Gleichzeitig wird ein Verhungern vermieden. Die Einführung von schmutzigen und sauberen Gabeln führt eine Rangfolge für hungernde Philosophen ein und benachteiligt automatisch Philosophen, die gerade eben gegessen haben.

In ihrer Analyse leiteten Chandy und Misra ein System von vorrangigen Ebenen für die Verteilung der Gabeln und ihren Status (sauber/schmutzig) ab. Sie zeigten das ihr System einen azyklischen Graphen beschreibt und alle Operationen in ihrem Protokoll den Graphen nicht zyklisch werden lassen können. Das garantiert das Verklemmungen nicht auftreten können. Allerdings kann ihr System dann zu einer Verklemmung führen, wenn das System mit einem perfekten symmetrischen Status initalisiert wird, z.B. wenn alle Philosophen die linke Gabel in ihrer Hand halten. In diesem Fall ist der Graph zyklisch und die Lösung kann eine

Verklemmung nicht mehr verhindern. Die Initialisierung des Systems, so dass die Philosophen mit den niedrigeren IDs schmutzige Gabeln halten, garantiert das der Graph azyklisch ist.

Implementierung in Java

Bei der Implementierung einer Lösung für die speisenden Philosophen in Java ist es möglich einen minimalistischen Ansatz zu wählen, der lediglich die Aufgabenstellung reflektiert. Als deutlich strukturierter erweist sich allerdings der objektorientierte Ansatz, bei dem alle realen Objekte direkt in der Lösung abgebildet werden.

Fork

Die wesentliche Problematik beim Philosophenproblem liegt darin, wie die Philosophen einzelne Gabeln für sich beanspruchen. Diese Operation muss stets atomar erfolgen und dafür eignet sich ein *ReentrantLock* sehr gut. Wir beginnen damit eine Klasse Fork (deut. Gabel) zu entwerfen.

[code xml:lang="java"]package eu.codeplanet.diningphilosophers; import java.util.concurrent.locks.ReentrantLock; /** * This is a single fork. Basically it is lock system. A ReentrantLock is owned * by the thread last successfully locking, but not yet unlocking it. A * thread invoking lock will return, successfully acquiring the lock, when * the lock is not owned by another thread. * * The constructor for this class accepts an optional fairness parameter. When * set true, under contention, locks favor granting access to the * longest-waiting thread. * * @author CodePlanet * @version 1.0.0, 06/12/2010 * @see CodePlanet */ public class Fork { /** * Initialize a single fork. * * @param owner if set to other than free sign, this fork has a owner */ public Fork(T owner) { _owner = owner; } /** * Picks up this fork by the owner. * * @param owner the owner * @return true, if successfully picked up */ public boolean pickUp(T owner) { boolean picked = false; try { _reentrantLock.lock(); // Check if fork is free if(hasOwner()) { picked = false; } else { _owner = owner; picked = true; } } finally { _reentrantLock.unlock(); // unlock } return picked; } /** * Drops this fork. */ public void drop() { try { // If the caller thread wants to drop this fork, we must first // be sure that this thread owns the lock. _reentrantLock.lock(); _owner = null; } finally { _reentrantLock.unlock(); } } /** * Returns true, if this seat has a owner. * * @return true if owner */ public boolean hasOwner() { return _owner != null ? true : false; } /** Reentrant lock for each fork */ private final ReentrantLock _reentrantLock = new ReentrantLock(true); /** A fork could have a owner, usually a philosopher */ private T _owner; }[/code]

Die Klasse Fork ist eine generische Klasse, die einen generischen Besitzer T aufweist. Der Besitzer einer Gabel ist in der Regel ein Philosoph, kann theoretisch allerdings beliebigen Typs sein. Eine Gabel lässt sich aufnehmen (engl. pick up) und wieder ablegen (engl. drop). Weiterhin kann überprüft werden, ob die Gabel bereits einen Besitzer hat.

Seat

Da das Restaurant für die Philosphen eine beliebige Zahl an Sitzplätzen bereitstellen kann und damit unsere Problemstellung vom klassischen Philosophenproblem leicht abweicht, besteht die nächste sinnvolle Aufgabe darin, die Sitzplätze ebenfalls zu definieren. Die Sitzplätze ähneln sehr stark den Gabeln, sie können gleichzeitig nur von einem Philosophen beschlagnahmt werden und haben entweder einen Besitzer oder keinen. Parallel zur Klasse Fork lässt sich nun die generische Klasse Seat implementieren.

[code xml:lang="java"]package eu.codeplanet.diningphilosophers; import java.util.concurrent.locks.ReentrantLock; /** * A single seat on the table. Each seat has its own ReentrantLock. * * @author CodePlanet * @version 1.0.0, 06/12/2010 * @see CodePlanet */ public class Seat { /** * Initialize a single seat. * * @param owner if sitting to other than free sign, this seat has a owner */ public Seat(T owner) { _owner = owner; } /** * Sit down. * * @param owner the owner, which wants to sit down. * @return true, if successfully sitting. */ public boolean sit(T owner) { boolean sitting = false; try { _reentrantLock.lock(); // Check if the seat is free if(hasOwner()) { sitting = false; } else { _owner = owner; sitting = true; } } finally { _reentrantLock.unlock(); // unlock } return sitting; } /** * Stand up. */ public void standUp() { try { // If the caller wants to stand up, we must first // be sure that the caller owns the lock. _reentrantLock.lock(); _owner = null; } finally { _reentrantLock.unlock(); } } /** * Returns true, if this seat has currently a owner. * * @return true if owner */ public boolean hasOwner() { return _owner != null ? true : false; } /** * Checks if the given object equals to the current owner of the seat. * * @param object the object to compare with * @return true if both objects not null and equal */ public boolean isOwner(T object) { boolean equal = false; if(_owner != null && object != null) { equal = _owner.equals(object) ? true : false; } return equal; } /** Reentrant lock for each seat */ private final ReentrantLock _reentrantLock = new ReentrantLock(true); /** A seat could have a owner, usually a philosopher */ private T _owner; }[/code]

Table

Das nächste Objekt ist der Tisch (engl. Table). Dazu realisieren wir eine generische Klasse mit dem Namen Table. Nachfolgend ist ein Auszug der Klasse abgebildet. Das Objekt wird mit *n* Sitzplätzen und einem PrintStream für die Ausgabe von Nachrichten initialisiert. Um einen konkurrierenden Zugriff auf die Listen zu ermöglichen, werden synchronisierte Listen verwendet. Anschließend wird der Tisch mit den Gabeln und mit den Sitzplätzen initialisiert.

[code xml:lang="java"]public class Table { /** * Initialze the table. * * @param nSeats number of seats at this table * @param ps a printstream for debug messages */ public Table(int nSeats, PrintStream ps) { _printStream = ps; _nSeats = nSeats; // This must be synchronized, since more than 1 threads // could access

and modify this lists. Probably this isn't // really necessary, because all threads only are reading them. _forksList = Collections.synchronizedList(new ArrayList(nSeats)); _seatsList = Collections.synchronizedList(new ArrayList(nSeats)); // Initialize the semaphore for the seats _seatSemaphore = new Semaphore(nSeats, true); // Initialize seats and forks for (int i = 0; i Philosopher

Die Klasse Philosopher ist die anspruchsvollste Klasse und beherbergt die eigentliche Logik der speisenden Philosophen. Um die größtmögliche Parallelität zu gewährleisten sollten alle Philosophen als Individuen unabhängig voneinander agieren können.

Um diese Individualität erreichen zu können, werden Philosophen als Threads realisiert. Die Klasse Philosopher erweitert dazu die Java-Klasse Thread. Im Konstruktor wird einem Philosophen ein Tisch, sowie eine unabhängige Thread-ID zugewiesen. Die Hauptmethode run, die von dem Thread ausgeführt wird, besteht im Wesentlichen aus den beiden Hauptbeschäftigungen eines Philosophen. Der Philosoph denkt die meiste Zeit und sobald er Hunger bekommt, geht er essen. In einer Endlosschleife werden diese beiden Tätigkeiten durchgeführt.

[code xml:lang="java"]/** * Main function of a philosopher. A philosopher thinks and eats. */ @Override public void run() { think(); // Philosopher always thinks at start while(_continue) { // Philosopher gets randomly hungry int number = Math.abs(new Random().nextInt() % (100)) + 0; if (number Die Realisierung von eat ist der schwierigere Teil. Ein Philosoph muss zunächst versuchen einen Sitzplatz am Tisch zu erhalten, bevor er versuchen kann Gabeln zu bekommen. Sitzplätze sind, wie bereits erläutert wurde atomare Objekte und können nur jeweils von einem Philosophen beschlagnahmt werden. Ein Semaphore am Tisch überwacht die Anzahl der freien Plätze und bildet eine faire Warteschlange für alle Philosophen, die einen Sitzplatz am Tisch ergattern möchten.

Sobald ein Philosoph einen Sitzplatz erhalten hat, versucht er abhängig von einer Zufallsvariable zunächst die linke oder rechte Gabel zu nehmen. Gelingt ihm das, versucht er die gegenüberliegende Gabel ebenfalls aufzuheben. Scheitert er bei dem Versuch, legt er die erste Gabel wieder ab und wartet eine kurze Zeitspanne, bevor er es erneut versucht. Auf diese Weise werden Zustände mit Verklemmungen verhindert, in denen jeder Philosoph eine Gabel für sich beansprucht und vergeblich darauf wartet das ein anderer Philosoph ihm die fehlende zweite Gabel überlässt.

Nachdem ein Philosoph erfolgreich beide Gabeln erhalten hat, kann ermit dem Mahl beginnen. Nach dem Mahl legt er die beiden Gabeln wieder ab und verlässt seinen Sitzplatz, um wieder in Ruhe denken zu können.

Master

Im Gegensatz zur Lösung von Chandy / Misra, verhindert der oben vorgestellte Programmcode das Verhungern von einzelnen Philosophen nicht. Um zu gewährleisten das alle Philosophen essen können und nicht verhungern, wird ein Master eingeführt, der in regelmäßigen Abständen kontrolliert, ob alle Philosophen im Durchschnitt gleich viel zu essen bekommen.

Der Master agiert ebenfalls als individueller Thread und überwacht das Essen am Tisch. Er bildet kontinuierlich Durchschnittswerte, aller von den Philosophen verspeisten Rationen und kontrolliert, ob jeder Philosoph im Durchschnitt liegt. Stellt der Master fest, dass ein Philosoph überproportional viele Rationen gegessen hat, sperrt er diesen Philosophen für eine bestimmte Zeit.

[code xml:lang="java"]// Check philosophers for eating violation for (int i = 0; i = _maxEatCount) { // Ban the bad philosopher _philosophers.get(i).setEatBan(true); } }[/code]

Auf diese Weise garantiert der Master, dass alle Philosophen auf Dauer ungefähr gleich viel zu essen bekommen.

Applet

Zum Abschluß dieses Tutorials finden Sie den vorgestellten Code in einem Java-Applet realisiert. Das Applet stellt eine minimalistische grafische Benutzeroberfläche für die Klassen zur Verfügung, so dass das Programm sofort getestet werden kann. Über einen PrintStream werden alle Statusmeldungen der speisenden Philosophen zum Textfeld des Applets umgeleitet. Der originale Code bleibt davon unberührt und kann ohne Änderungen direkt in einer Konsolenanwendung oder einer normalen Java-Anwendung weiterverwendet werden.

Die Einstellungen in dem Applet sind minimal gehalten und nutzen nur die wesentlichen Optionen. Weitere Optionen, wie Bannzeit, Hungerwert, bevorzugter Sitzplatz eines Philosophen usw. werden von den Klassen unterstützt und können bei Bedarf genutzt werden. Die Anzahl der Philosophen und Sitzplätze ist auf 10 begrenzt.

Bei nur einem Sitz und mehreren Philosophen lässt sich erkennen das kein Philosoph essen kann, da bei nur einem Sitz auch nur eine Gabel zur Verfügung steht. Ein Philosoph wartet maximal 200 Zeiteinheiten, bevor er aufgibt und seinen Sitzplatz wieder verlässt.