Aktorenmodell am Beispiel Erlang

Um die Leistung von Mehrprozessorkernen nutzen zu können, müssen Programme so geschrieben sein, dass die verfügbaren Prozessoren auch beschäftigt werden. Die Entwicklung solcher Programme ist jedoch eine grosse Herausforderung. Erleichterung versprechen Konzepte wie das Aktorenmodell. In diesem Artikel wird das Aktorenmodell im Kontext der Programmiersprache Erlang vorgestellt und es wird aufgezeigt, dass auch dieses Modell seine Tücken hat.

Dominik Gruntz | dominik.gruntz@fhnw.ch

Jahr für Jahr steigerte sich in der Vergangenheit die Leistung der PCs, in dem die Taktrate erhöht wurde. Die Taktrate von auf aktuellen Prozessortechnologien basierenden Rechnern kann aus physikalischen Gründen nicht mehr gross wachsen. Die Leistung wird jedoch weiterhin zunehmen, indem Rechner mit mehreren Kernen bestückt werden. Ein Notebook, das heute bei einem Händler gekauft wird, ist typischerweise mit einem Doppelkernprozessor ausgestattet, aber es können auch Rechner mit vier, sechs oder mehr Kernen gekauft werden. Es ist zu erwarten, dass die Anzahl der Kerne pro Prozessor in den nächsten Jahren einen ähnlichen Verlauf nimmt wie bisher die Megahertz-Zahlen.

Damit die Software diese Leistung auch nutzen kann, muss sie parallel auf diesen Kernen ausgeführt werden können. Multicore-Rechner sind nutzlos ohne entsprechende Programme. Eine single-threaded Anwendung kann auf einem Rechner mit n Kernen nur $1/n$ der Leistung ausnützen. Die Devise, dass die Software mit der Leistungssteigerung der Prozessoren automatisch schneller wird, gilt nicht mehr [Sut05].

Programme müssen also entsprechend parallelisiert werden und die Software muss so programmiert sein, dass sie mit der Anzahl der Kerne skaliert. Problematisch ist dabei die korrekte Synchronisation mehrer Threads bei gemeinsamen Datenzugriffen. Die klassischen Locking-Mechanismen bergen viele Fallstricke. In Java sind zumindest die Garantien, welche die VM beim Zugriff auf den Hauptspeicher bietet, im *Java Memory Model* [JLS05] definiert. Bei anderen Sprachen fehlen solche Garantien.

Die Suche nach dem optimalen Programmiermodell für parallele Hardware bringt Programmiersprachen in den Fokus, die bisher eher als Exoten galten. Zu den interessanteren Modellen gehören *Aktoren,* wie sie von *Erlang*, *Scala* und *Axum* angeboten werden, sowie *Transactional Memory*, wie es *Clojure* und *Scala* unterstützen. Im Studiengang Informatik der FHNW führen wir das Modul *Concurrent Programming* durch, in welchem wir diese verschiedenen Modelle den Studierenden vermitteln.

In diesem Artikel konzentrieren wir uns auf das Aktorenmodell und stellen dieses im Kontext von Erlang vor. Dazu werden wir kurz in die Programmiersprache Erlang einführen und zeigen, dass die Implementierung eines thread-sicheren und blockierenden Stacks in Erlang trivial ist. Dass jedoch auch Erlang seine Fallstricke hat, sehen wir, wenn wir diese Stack-Implementierung so erweitern, dass die *pop*-Operation mit einem Timeout abbricht.

Erlang

Erlang ist eine Sprache, die 1984 von einem Team um Joe Armstrong bei den *Ericsson Computer Science Labs* speziell für die Programmierung paralleler und robuster Systeme entwickelt wurde. Erlang war ursprünglich in *Prolog* implementiert, daher erinnert die Syntax von Erlang stark an Prolog. Seit 1998 ist Erlang unter einer Open Source Lizenz frei verfügbar.

Erlang ist dynamisch typisiert und wird auf einer eigenen VM ausgeführt. Erlang-Programme bestehen aus individuellen Prozessen, welche nicht auf gemeinsamen Speicher zugreifen, sondern nur über asynchronen Nachrichtenaustausch miteinander kommunizieren können. Erlang-Prozesse sind sehr leichtgewichtig und werden durch das Erlang-Laufzeitsystem kontrolliert und nicht durch das darunterliegende Betriebssystem. Aus diesem Grund können „beliebig“ viele Prozesse aktiv sein (per Default können maximal 32768 Prozesse gleichzeitig aktiv sein, diese Grenze kann jedoch bei Programmstart auf 268435456 erhöht werden). Da die Nebenläufigkeit inhärent in die Sprache integriert worden ist, wird sie von Armstrong auch gerne als *concurrency-oriented programming language* (COPL) bezeichnet [Arm02].

Die Strukturierungseinheit in Erlang sind Module. Ein Modul kann mehrere Funktionen exportieren. Nicht exportierte Funktionen sind privat. Eine Funktion kann mit mehreren Klauseln definiert werden, die je durch ein Semikolon getrennt werden. Jede Klausel besteht aus einem Muster und auszuführenden Aktionen. Bei einem Funk-

```
-module(math).
-export([faculty/1, area/1]).

faculty (N) when N > 0 ->
   N * fac(N - 1);
faculty (0) -> 1.

area( {rectangle, X, Y} ) -> X*Y;
area( {circle, R} ) ->
   3.1415926535 * R * R;
area( _ ) -> false.
```

Listing 1: Modul math

tionsaufruf werden die Muster der Klauseln in der Deklarationsreihenfolge geprüft und die erste passende Klausel wird ausgeführt. Die im Muster auftretenden Variablen werden dabei gebunden. Die Klauseln können zusätzlich mit einem Wächter (Guard) versehen werden. Innerhalb einer Klausel trennen Kommas die einzelnen Anweisungen. Ein Punkt bezeichnet das Ende einer Funktionsdefinition. Das in Listing 1 abgedruckte Modul *math* definiert die Funktionen *faculty* und *area*, welche aus jedem anderen Modul (oder aus dem Erlang-Interpreter) mit *math:faculty* bzw. *math:area* aufgerufen werden können.

In Erlang sind alle Objekte unveränderbar (immutable). Variablen können nur einmal gebunden und danach nicht mehr verändert werden. Eine Zuweisung „X := X+1“ ist in Erlang damit nicht möglich. Variablen beginnen in Erlang immer mit Grossbuchstaben. Atome sind Literale, die mit Kleinbuchstaben beginnen. Als strukturierte Datentypen bietet Erlang Listen und Tupel an. Listen werden mit eckigen Klammern und Tupel mit geschweiften Klammern notiert. Sowohl Listen als auch Tupel können Objekte unterschiedlicher Typen enthalten. Oft wird (wie in Listing 1) ein Atom zur Identifikation eines Tupels verwendet.

Erlang unterstützt nebenläufige Programmierung durch Prozesse, welche untereinander ausschliesslich über asynchrone Nachrichten kommunizieren können. Jeder Prozess besitzt eine Mailbox, in welcher die eingegangenen und noch nicht verarbeiteten Meldungen abgelegt werden. Diese Meldungen werden vom Prozess sequentiell abgearbeitet. Dabei können neue Prozesse gestartet und weitere Meldungen verschickt werden.

Das Programmiermodell von Erlang ist damit weitgehend identisch mit dem von Gul Agha vorgeschlagenen Aktorenmodell [Agh86]. Ein Aktor ist ein aktives Objekt, dessen Code von genau einem Prozess ausgeführt wird. Ein Aktor verwaltet seinen Zustand und kommuniziert mit anderen Aktoren nur über asynchronen Nachrichtenaustausch. In Erlang ist jeder Prozess ein Aktor.

Ein neuer Aktor wird in Erlang mit der Funktion *spawn(Fun)* erzeugt und gestartet. Der Aktor führt dabei die Funktion *Fun* aus, welche als Parameter übergeben wird. Das Resultat von *spawn()* ist die ID des gestarteten Aktor-Prozesses.

Meldungen werden mit dem !-Operator an einen Aktor geschickt. Wird die Anweisung *Pid ! Msg* ausgeführt, so wird die Meldung *Msg* in die Mailbox des Aktors mit der Prozess-ID *Pid* gelegt.

Mit der *receive* Anweisung können mit Hilfe von Pattern-Matching Meldungen aus der Mailbox ausgelesen werden. Die Anweisung

```
receive
  Pattern1 -> Actions1;
  Pattern2 -> Actions2
end
```

verarbeitet die Meldungen aus der Mailbox. Falls die älteste Meldung dem Muster *Pattern1* entspricht, werden die Anweisungen *Actions1* ausgeführt, andernfalls wird geprüft, ob die Meldung dem Muster *Pattern2* entspricht. Falls dies der Fall ist, werden die Anweisungen *Actions2* ausgeführt. Wenn kein Muster passt, so wird die Meldung zurückgestellt und es wird die nächste

```
-module(stack).
-export([start/0, pop/0, push/1]).

start() ->
  register(stack, spawn(fun() -> stack([]) end)).

push(Element) ->
  stack ! {push, Element}, ok.

pop() ->
  stack ! {pop, self()},
  receive R -> R end.

stack([]) ->
  receive
    {push, Element} -> stack([Element])
  end;
stack(List) ->
  receive
    {push, Element} -> stack([Element|List]);
    {pop, Pid} -> [H|T] = List, Pid ! H, stack(T)
  end.
```

Listing 2: Blockierender Stack

Meldung in der Mailbox bearbeitet. Sind keine weiteren Meldungen zur Ausführung bereit, so wird auf eine neue Meldung gewartet. Falls die Mailbox leer ist, dann blockiert die *receive* Anweisung ebenfalls.

Blockierender Stack

Im Folgenden zeigen wir eine einfache Implementierung eines Stacks. Das Modul *stack* in Listing 2 implementiert einen Stack, auf den mit den Funktionen *push()* und *pop()* zugegriffen werden kann. Mit der Funktion *start()* wird der Prozess gestartet, welcher den Stack kontrolliert. Dabei wird ein neuer Erlang-Prozess erzeugt, der die Funktion *stack([])* mit der leeren Liste als Argument ausführt. Diese Funktion verarbeitet die an den Stack geschickten Meldungen. Man könnte diese Datenstruktur leicht so erweitern, dass sich gleichzeitig mehrere Stacks verwenden liessen. Dazu müsste nur die Funktion *start()* die Prozess-ID des generierten Prozesses zurückgeben, und diese müsste dann bei den Funktionen *push()* und *pop()* als zusätzlichen Parameter übergeben werden.

Der Aktor akzeptiert als Meldungen die Tupel *{push, Element}* und *{pop, Pid}*. Dieses Meldungsprotokoll wird jedoch in den Funktionen *push()* und *pop()* gekapselt, d.h. der Client kann aus einem beliebigen Aktor die Funktionen *push()* und *pop()* aufrufen und damit auf den Stack zugreifen.

Die Funktion *push()* ist asynchron implementiert, d.h. ein Aufruf terminiert sofort nachdem die Meldung *{push, Element}* in der Mailbox des Stacks abgelegt worden ist. Erlang garantiert, dass zwei Meldungen, welche ein Aktor A_1 hintereinander an einen Aktor A_2 schickt, dann auch in dieser Reihenfolge in der Mailbox von Aktor A_2 abgelegt werden.

Die Funktion *pop()* ist synchron, da hier ein Resultat erwartet wird. Eine synchrone Kommunikation zwischen zwei Aktoren wird dadurch erreicht, dass der aufrufende Aktor auf eine Antwort wartet. Dazu gibt er beim Aufruf seine eigene Prozess-ID *self()* als Teil der Meldung mit.

Die Funktion *stack()* implementiert das Verhalten des Stack-Aktors. In dieser Funktion werden die Meldungen aus der Mailbox abgearbeitet. Im Argument der Funktion *stack* sind die Elemente des Stacks in einer Liste abgelegt. Bei den rekursiven Aufrufen in dieser Funktion handelt es sich um endrekursive Aufrufe, bei welchen kein neuer Stackframe erzeugt wird; d.h. diese rekursiven Aufrufe entsprechen eigentlich einer Schleife.

Das interessanteste an dieser Stack-Implementierung jedoch ist, dass der Stack nicht nur thread-sicher ist, sondern es handelt sich auch um einen blockierenden Stack. Wird die Funktion *pop()* aufgerufen wenn der Stack leer ist, dann blockiert dieser Aufruf, d.h. die entsprechende Meldung bleibt in der Mailbox des Stack-Aktors liegen. Sie wird erst bearbeitet, wenn wieder Elemente im Stack enthalten sind. Sobald eine *push*-Meldung in der Mailbox abgelegt wird, wird nach dem Bearbeiten dieser *push*-Meldung die älteste pendente *pop*-Meldung abgearbeitet. Die *receive*-Anweisung versucht also immer die zur ältesten Meldung passenden Aktionen auszuführen.

Timeouts

Für die Benutzer eines blockierenden Stacks ist es hilfreich, wenn neben der für unbestimmte Zeit blockierenden Operation *pop()* noch eine weitere Operation zur Verfügung steht, welche nach einer bestimmten Wartezeit mit einer entsprechenden

```
pop(Timeout) ->
  stack ! {pop, self()},
  receive R -> R
  after Timeout ->
    stack ! {cancel_pop, self()},
    receive R -> R end                      % this may either be the popped element
                                            % or the token "timeout"
  end.

stack([]) ->
  receive
    {push, Element} -> stack([Element]);
    {cancel_pop, Pid} -> cancel_pop(Pid), stack([])
  end;
stack(List) ->
  receive
    {push, Element} -> stack([Element|List]);
    {pop, Pid} -> [H|T] = List, Pid ! H, stack(T);
    {cancel_pop, Pid} -> cancel_pop(Pid), stack(List)
  end.

cancel_pop(From) ->
  receive {pop, From} -> From ! timeout     % remove the message {pop, From} from its
  after 0 -> nothing                        % own mailbox or do nothing if
                                            % this message was already handled
  end.
```

Listing 3: Erweiterung der Stack-Implementierung für die neue Funktion pop(Timeout)

Meldung terminiert. Dieses Verhalten soll in der Funktion *pop(Timeout)* implementiert werden, bei welcher ein Timeout spezifiziert werden kann. Erlang unterstützt Timeouts mit einer speziellen *after*-Klausel in der *receive*-Anweisung. Man könnte die *pop*-Funktion damit wie folgt erweitern:

```
pop(Timeout) ->
  stack ! {pop, self()},
  receive R -> R
  after Timeout -> timeout
  end.
```

Die *receive*-Anweisung (und damit die *pop*-Funktion) terminieren so nach *Timeout* Millisekunden mit dem Resultat *timeout*, wenn nicht vorher eine Antwort vom Stack-Aktor ausgeliefert worden ist. Das Problem jedoch ist, dass die *pop*-Meldung bereits an den Stack-Aktor geschickt worden ist und dass damit das angeforderte Element irgendwann in der Mailbox des Aufrufers landet.

Ein Timeout in der *receive*-Anweisung des Stack-Aktors einzubauen wäre denkbar, aber da solche zeitlich beschränkte *pop*-Anfragen von unterschiedlichen Aktoren mit verschiedenen Timeouts eingehen können, wird die Buchhaltung etwas aufwändig. Wir lösen das Problem daher damit, dass wir die Meldung, die wir an den Aktor geschickt haben, mit einer weiteren Meldung wieder zurückziehen. Es ist jedoch zu beachten, dass der Server die Antwort bereits verschickt haben könnte, bevor er die Rückzugsmeldung erhält. In diesem Fall muss die Rückzugsmeldung vom Stack-Aktor ignoriert werden. Die Implementierung der Funktion *pop(Timeout)* und die Anpassungen in der vom Aktor-Prozess ausgeführten Funktion *stack(List)* sind in Listing 3 abgebildet.

Läuft also der Timeout in der Funktion *pop()* ab, so wird die Meldung *{cancel_pop, self()}* an den Stack-Aktor geschickt. Diese Meldung muss sowohl im leeren als auch im nicht-leeren Stack-Zustand behandelt werden, denn der Stack kann in beiden Zuständen sein, wenn er diese Meldung empfängt. Der Stack-Aktor versucht, die ursprüngliche *pop*-Meldung aus der Mailbox zu entfernen. Falls diese Meldung nicht mehr in der Mailbox vorhanden ist, dann ist sie bereits behandelt worden und dem anfragenden Aktor ist ein Element des Stacks zugestellt worden. Andernfalls wird die Meldung aus der Mailbox entfernt und dem Klienten wird das Token *timeout* geschickt, welches dieser als Resultat der Funktion *pop()* zurückgibt. Der Client erhält also auf jeden Fall ein Resultat auf die *cancel_pop*-Anfrage.

Korrektheit

Leider ist die im letzten Abschnitt vorgestellte Implementierung nicht korrekt. Nehmen wir an, dass ein Aktor A_1 die Funktion *pop(1000)* ausführt. Der Stack sei leer, daher sendet der Aktor A_1 nach 1000 ms die Meldung *{cancel_pop, A1}* an den Stack-Aktor. Gerade bevor diese Meldung beim Stack ankommt, wird von einem anderen Prozess A_2 ein Element auf den Stack gelegt, d.h. der Stack sendet an den Aktor A_1 das soeben platzierte Stack-Element zurück. In der Mailbox des Stack-Aktors liegt vom Aktor A_1 nur noch die Meldung *{cancel_pop, A1}*. Möglicherweise liegen in der Mailbox noch weitere Meldungen anderer Aktoren. Der Stack-Aktor wird diese Meldungen nun sequentiell in der Reihenfolge ihres Einganges abarbeiten; gleichzeitig können andere Aktoren jederzeit weitere Meldungen in der Mailbox des Stack-Aktors ablegen. Wir nehmen nun an, dass A_1 nach Erhalt des Resultats sogleich ein neues Element anfordert. Er ruft dazu die blockierende Funktion *pop()* auf (also die Version ohne Timeout) und sendet damit die Meldung *{pop, A1}* an den Stack. Wenn der Stack nun die *cancel_pop*-Meldung bearbeitet, kann es sein, dass die zweite *{pop, A1}* Anfrage bereits in der Mailbox liegt. Der Stack-Aktor wird diese entfernen und an A_1 als Antwort das Token *timeout* zurückschicken. Der zweite *pop*-Aufruf von A_1 wird also mit einem *timeout* beantwortet (was signalisiert, dass die *pop*-Anfrage gelöscht wurde) und dies ist ein fehlerhaftes Verhalten. Der Stack-Aktor hätte die *cancel_pop*-Meldung verwerfen müssen, denn das zuerst angeforderte Element ist bereits ausgeliefert worden, und der zweite *pop()*-Aufruf von A_1 hätte blockieren müssen bis ein Element vorhanden ist.

Dieses Beispiel zeigt, dass es auch bei parallelen Programmen, die mit dem Aktorenmodell von Erlang realisiert sind, nicht einfach ist, die Korrektheit nachzuweisen. Sobald man jedoch ein Problem identifiziert hat, kann dieses leicht behoben werden. In obigem Beispiel könnte man die *pop*-Anfragen mit einer eindeutigen Nummer versehen. Die Funktion *erlang:make_ref()* erzeugt eine eindeutige Referenz – beinahe eindeutig, denn nach 2^{82} Aufrufen wiederholen sich die generierten Referenzen, aber solche Referenzen sind für unsere Zwecke eindeutig genug. Der Aktor muss entsprechend angepasst werden, damit nur noch die Anfrage mit der entsprechenden Referenz entfernt wird. Die geänderte Funktion *pop(Timeout)* sieht dann wie folgt aus:

```
pop(Timeout) ->
  MsgID = erlang:make_ref(),
  stack ! {pop, self(), MsgID},
  receive R -> R
  after Timeout ->
    stack ! {cancel_pop, self(), MsgID},
    receive R -> R end
  end.
```

Eine andere Lösung wäre, dass in der Funktion *pop(Timeout)* gewartet wird, bis der Stack-Aktor die *cancel_pop*-Anfrage bearbeitet hat. Nehmen wir an, dass dazu der Stack-Aktor auf eine *can-*

cel_pop-Anfrage entweder das Token *pop_deleted* zurück gibt, falls eine *pop*-Anfrage vom entsprechenden Klienten gefunden und gelöscht wurde, oder das Token *pop_notfound,* falls die *pop*-Anfrage bereits beantwortet worden ist. Die Funktion *pop(Timeout)* kann dann wie folgt implementiert werden:

```
pop(Timeout) ->
  stack ! {pop, self()},
  receive R -> R
  after Timeout ->
    stack ! {cancel_pop, self()},
    receive
      pop_deleted -> timeout;
% pop-request was removed by stack actor
      pop_notfound ->
  % pop-request was not found by stack
actor, i.e. element has
       receive R -> R end
% already been sent to client process
    end
  end.
```

Nachdem die Meldung *{cancel_pop, self()}* an den Stack-Aktor geschickt worden ist, kann es sein, dass der Stack als Antwort auf die ursprüngliche *pop*-Anfrage ein Element an den Client zurückschickt. Dieses bleibt dann jedoch vorerst in der Mailbox des Klienten liegen, denn die *receive*-Anweisung, die nach dem Versand der *cancel_pop*-Melung ausgeführt wird, wartet explizit, bis der Stack-Aktor eines der beiden Atome *pop_deleted* oder *pop_notfound* zurücksendet, erst dann wird allenfalls das vorab eingegangene Element aus der Mailbox ausgelesen und an den Aufrufer zurückgegeben. Mit dieser Lösung wird verhindert, dass ein Prozess bereits eine weitere *pop*-Meldung an den Stack-Aktor schickt, bevor dieser die *cancel_pop* Meldung bearbeitet hat. Diese Lösung ist allerdings nur dann korrekt, wenn der Prozess die vom Modul *stack* bereitgestellten Funktionen verwendet und nicht direkt Meldungen an den Stack-Aktor sendet.

Fazit

Die Entwicklung von korrekten und effizienten parallelen Programmen ist schwierig. Es liegt dem sequentiell denkenden Menschen nicht, sich die Abläufe von parallel arbeitenden Prozessen vorzustellen. Das Aktoren-Modell eliminiert viele Fehlerquellen, da Aktoren isoliert sind und nur über asynchrone Meldungsprotokolle miteinander kommunizieren. Die Schwäche von Erlang sind aber gerade diese Meldungsprotokolle. Eine saubere Spezifikation dieser Protokolle ist zwingend nötig. Im Zweifelsfall verwendet man besser synchrone Protokolle (womit sich jedoch die Gefahr von Verklemmungen wieder erhöht).

Das Aktoren-Modell ist sicher eine Hilfe beim Programmieren von parallelen Programmen, aber auch dieses Modell garantiert nicht automatisch, dass die Programme fehlerfrei sind. Leider gilt auch hier, dass sich solche Probleme kaum durch Tests finden lassen. Die Befolgung von etablierten Prozess-Entwurfsmustern hilft Fehler zu vermeiden.

Links

Axum	http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx
Erlang	http://www.erlang.org
Clojure	http://clojure.org
Scala	http://www.scala-lang.org

Modul Concurrent Programming
http://web.fhnw.ch/plattformen/conpr/

Referenzen

[Agh86] Gul Agha. Actors: Actors: a model of concurrent computation in distributed systems, MIT Press, 1986.

[Arm10] Joe Armstrong. Erlang, Communications of the ACM, Vol. 53, No. 9, p. 68-75, Sept 2010.

[Arm02] Joe Armstrong. Concurrency-oriented programming in Erlang, Invited Talk at the Lightweight Languages Workshop (Cambridge MA, Nov. 9, 2002).

[Sut05] Herb Sutter. The free lunch is over, Dr. Dobb's Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm

[JLS05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. The Java Language Specification, 3rd Edition, Section 17.4: Memory Model, Addison Wesley, 2005. http://java.sun.com/docs/books/jls/third_edition/html/memory.html#17.4