Einführung in die Informatik 2

Prof. Dr. Seidl, J. Kranz, N. Hartmann, J. Brunner — WS 2016/17

Übungsblatt 14 — Abgabefrist: 13.2.2017

Aufgabe 14.1 (P) Nebenläufige Kamele

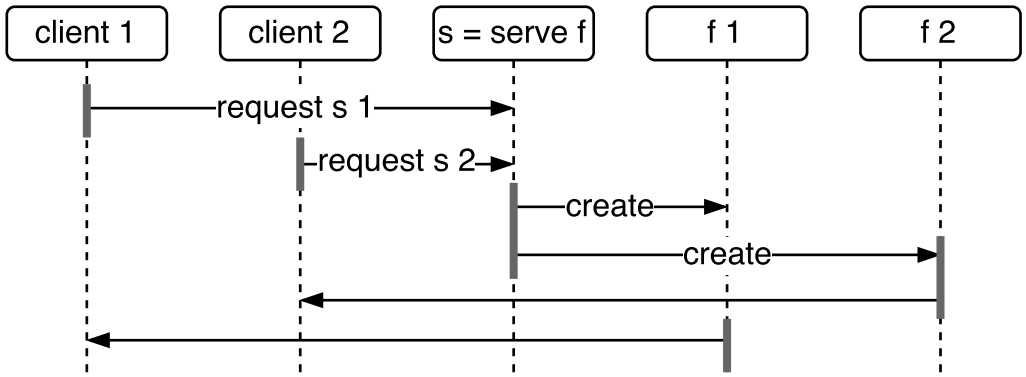
Implementieren Sie das Modul

```
module Server : sig
  type ('a, 'b) t
  val serve : ('a -> 'b) -> ('a, 'b) t
  val request : ('a, 'b) t -> 'a -> 'b
end
```

wobei

- `('a, 'b) t` den Typ eines Servers darstellt, der Anfragen vom Typ `'a` entgegennimmt und Ergebnisse vom Typ `'b` liefert,
- `serve f` einen Thread startet, der Anfragen mit der Funktion `f` beantwortet. Für die Bearbeitung einer Anfrage soll dazu jeweils wieder ein eigener Thread gestartet werden und
- `request s a` dem Server `s` eine Anfrage `a` schickt und auf dessen Antwort wartet.

Achten Sie darauf, dass Anfragen transaktionell bearbeitet werden – das Ergebnis also invariant gegenüber der Auswertungsreihenfolge von Threads ist. Werden z.B. `request s 1` und `request s 2` von unterschiedlichen Threads aus aufgerufen, so muss sichergestellt werden, dass jeder Request die richtige Antwort bekommt.



Lösungsvorschlag 14.1

```
open Thread open Event
type ('a, 'b) t = ('a * 'b channel) channel

let serve f =
  let c = new_channel () in
  let rec task () =
```

```
    let a,r = sync (receive c) in
    let _ = create (fun () -> sync (send r (f a))) () in
    task ()
  in
  let _ = create task () in
  c

let request s a =
  let c = new_channel () in
  sync (send s (a, c));
  sync (receive c)
```

Aufgabe 14.2 (P) Parallele Vektor-Matrix-Multiplikation

Die Multiplikation eines Zeilenvektors und einer Matrix sei definiert wie folgt:

$$(x_1\ x_2\ \ldots\ x_n) * \begin{pmatrix} y_{1,1} & y_{2,1} & \ldots \\ y_{1,2} & y_{2,2} & \ldots \\ \vdots & \vdots & \vdots \\ y_{1,n} & y_{2,n} & \ldots \end{pmatrix} = \begin{pmatrix} y_{1,1} * x_1 + y_{1,2} * x_2 + \ldots + y_{1,n} * x_n \\ y_{2,1} * x_1 + y_{2,2} * x_2 + \ldots + y_{2,n} * x_n \\ \vdots \end{pmatrix}^T$$

Eine Matrix wird in dieser Aufgabe als Liste von Zeilen repräsentiert. Jede Zeile wiederum ist eine Liste von Integern. Einen Vektor repräsentieren wir ebenfalls als Liste von Integern. Wir möchten die Multiplikation eines Vektors und einer Zeile parallelisieren und wie folgt auf Threads verteilen.

- Der Transposer-Thread erwartet beim Start einen Ausgabe-Channel und eine Matrix. Er transponiert die Matrix, wobei er jede neue Zeile über den Channel an den Master schickt. Der Transposer sendet `None` an den Master, wenn die gesamte Matrix transponiert wurde.
- Der Multiplikator-Thread erwartet beim Start einen Vektor `v`, sowie einen Eingabe- und Ausgabe-Channel. Er empfängt wiederholt Vektoren aus dem Eingabe-Channel, die er anschließend mit `v` elementweise multipliziert. Das Ergebnis sendet er dabei über den Ausgabe-Channel an den Master. Empfängt der Multiplikator `None` am Eingabe-Channel, beendet er sich.
- Der Addierer-Thread erwartet beim Start einen Eingabe- und Ausgabe-Channel. Er empfängt wiederholt Vektoren aus dem Eingabe-Channel, bei denen er anschließend ihre Elemente aufaddiert. Das Ergebnis sendet er dabei über den Ausgabe-Channel an den Master. Empfängt der Addierer `None` am Eingabe-Channel, beendet er sich.
- Der Master-Thread koordiniert die Kommunikation. Er empfängt jeweils eine Nachricht von einem der obigen Worker-Threads. Aus der Nachricht ergeben sich neue Arbeitsaufträge oder es ergibt sich ein neues Element des Ergebnisses.

Komplettieren Sie den Master-Thread an der angegebenen Stelle. Verwenden Sie die `select`-Funktion, um dazu eine Nachricht von demjenigen Worker zu empfangen, der als nächstes einen Arbeitsschritt abgeschlossen hat. Achten Sie darauf, keine Verklemmungen zu erzeugen sowie ein deterministisch richtiges Ergebnis zu erzeugen.

Lösungsvorschlag 14.2

Die Lösung befindet sich in der Datei `ocaml/p14_sol.ml`.

Allgemeine Hinweise zur Hausaufgabenabgabe

Die Hausaufgabenabgabe bezüglich der Programmieraufgaben dieses Blattes erfolgt über das Ocaml-Abgabesystem. Sie erreichen es unter `https://vmnipkow3.in.tum.de`. Sie können hier Ihre Abgaben einstellen und erhalten Feedback, welche Tests zu welchen Aufgaben erfolgreich durchlaufen. Sie können Ihre Abgabe beliebig oft testen lassen. Bitte beachten Sie, dass Sie Ihre Abgabe stets als **eine einzige UTF8-kodierte Textdatei mit dem Namen *ha14.ml* hochladen müssen**. Die hochgeladene Datei muss ferner den Signaturen in *ha14.mli* entsprechen. Die Hausaufgabenabgabe bezüglich der Malaufgaben erfolgt persönlich bei der Übungsleitung.

Aufgabe 14.3 (H) Ticket-Server mit Threads [20 Bonuspunkte]

In dieser Aufgabe geht es darum, einen Ticket-Server mittels Threads zu implementieren. Grundlegend soll die Kommunikation wie folgt ablaufen:

1. Ein Client verbindet sich per TCP/IP zum Server.
2. Nachdem die Verbindung aufgebaut wurde, sendet der Client einen Befehl an den Server. Es stehen dabei folgende Befehle zur Verfügung:
   - `Register`: Anmeldung eines Nutzers (mittels Vor- und Nachname)
   - `Status`: Abfrage des Status einer vorhandenen Anmeldung
   - `Deregister`: Abmeldung eines Nutzers (mittels Vor- und Nachname)
3. Der Server antwortet dem Client daraufhin wie folgt.
   - `Success`: Erfolgreiche Anmeldung oder Statusabfrage; bei dieser Antwort schickt der Server auch das reservierte Ticket mit.
   - `DeregisterSuccess`: Erfolgreiche Abmeldung
   - `InvalidRequest`: Die Anfrage konnte nicht verarbeitet werden (siehe unten bzgl. des verwendeten Serialisierungsverfahrens)
   - `UnknownUser`: Unbekannter Nutzer (Statusabfrage, Abmeldung)
   - `AlreadyRegistered`: Der Nutzer ist bereits angemeldet; doppelte Anmeldungen sind nicht möglich.
   - `NoMoreTickets`: Alle Tickets sind vergeben; die Anmeldung war daher nicht erfolgreich.
   - Der Server schließt die Verbindung.

Der Server soll der Lage sein, mehrere Verbindungen gleichzeitig anzunehmen. Um dies zu ermöglichen, müssen mehrere Threads die Klienten verwalten. Es soll hier folgende Architektur verwendet werden:

- Der initiale Thread ist der *Master*. Er startet den *Server*-Thread und wartet anschließend auf Nachrichten der Clients, die er über einen *Channel* erhält. Er verarbeitet die Anfragen und generiert jeweils eine Antwort. Es wird also jeweils nur eine Anfrage zur gleichen Zeit verarbeitet.

- Der *Server*-Thread wartet auf neue Clients. Sobald ein neuer Client sich mit dem Server verbindet, wird ein *Client*-Thread gestartet, der die Kommunikation zwischen dem Client und dem Master-Thread koordiniert. Der Server-Thread dagegen fährt damit fort, auf weitere Clients zu warten.
- Die *Client*-Threads warten zunächst auf die Anfrage der Clients. Sobald diese empfangen wurde, muss sie deserialisiert und an den Master weitergeleitet werden. Sobald dieser die Anfrage beantwortet hat, serialisiert der jeweilige Client-Thread die Antwort und leitet sie an den verbundenen Client weiter. Anschließend schließt er die Verbindung und beendet sich.

Die Daten müssen serialisiert werden, um über das Netzwerk verschickt zu werden. Empfangene Daten wiederum müssen deserialisiert werden, um sie in einen Wert eines Ocaml-Typen zu verwandeln. Wir werden hier sogenannte *S-Expressions*[1,2] zur Serialisierung verwenden. Damit Werte des Typen `request` bzw. `response` automatisch umgewandelt werden können, ist das Paket `ppx_sexp_conv` erforderlich (siehe dazu auch das Wiki in Moodle). Die Typen der Angabe sind schon passend annotiert, sodass sie eine Request-S-Expression `s` im String-Format durch

```
request_of_sexp (Sexplib.Sexp.of_string s)
```

in einen Ocaml-Werte umwandeln können. Umgekehrt können Sie einen Ocaml-Wert `w` des Typen `response` durch

```
Sexplib.Sexp.to_string (sexp_of_response w)
```

in eine S-Expression im String-Format umwandeln. Ein Request bzw. eine Response wird als S-Expression im String-Format gefolgt von einem Zeilenumbruch (’\n’) gesendet.

Ihr Programm soll durch die Funktion

```
val master : Unix.inet_addr -> int -> int -> 'a
```

gestartet werden können. Sie erwartet zunächst eine lokale Adresse und einen Port für den Server. Der dritte Parameter enthält die Anzahl der Tickets, die insgesamt zur Verfügung stehen.

Hinweis: Die Angabe dieser Aufgabe und die erwartete Lösung ist (vermutlich) nicht portabel und sollte daher unter Linux entwickelt werden. Lösungen, die Windows-spezifische APIs (oder andere) nutzen, können leider nicht bewertet werden. In der Rechnerhalle des MI-Gebäudes bieten wir Rechner an, auf denen Sie Ihre Hausaufgaben lösen können.

Lösungsvorschlag 14.3

Die Lösung befindet sich in der Datei `ocaml/ha14_sol.ml`.

Korrekturhinweise:

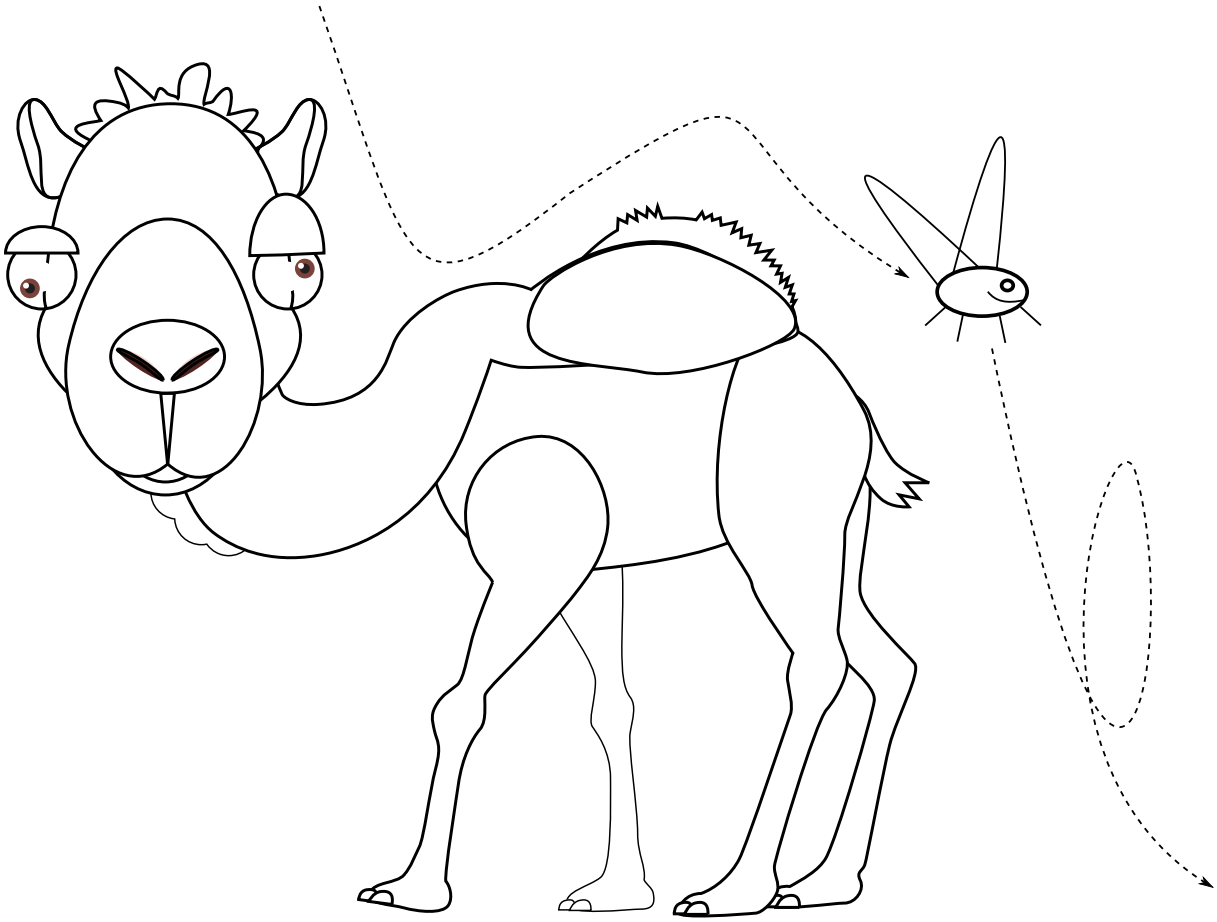
- Nur der letzte Test prüft effektiv, ob Threads verwendet werden. Der Grund dafür ist, dass verschachtelte Anfragen komplexer zu debuggen sind. Bei der Bewertung dürfen allerdings maximal 3 Punkte insgesamt vergeben werden, wenn die Aufgabe gänzlich ohne Threads gelöst wurde.
- Die Tests haben auf dem Abgabesystem nicht funktioniert. Die Tutoren müssen daher die Abgaben der Studenten herunterladen und lokal testen.

[1] `https://github.com/janestreet/sexplib`
[2] `https://realworldocaml.org/v1/en/html/data-serialization-with-s-expressions.html`

Aufgabe 14.4 (H) Besser ein Ende mit Kamel, als Kamele ohne Ende

[5 Bonuspunkte]

In dieser Aufgabe geht es darum, die folgende Kamel-Szene möglichst ansprechend auszumalen. Nutzen Sie hierzu eine große Vielzahl verschiedener Buntstifte unterschiedlicher Farben sowie ausreichend Glitzer. Achten Sie darauf, einzelne Module des Kamels gleichmäßig zu färben, sodass ein künstlerisch anspruchsvolles Gesamtbild entsteht.



Hinweis: Achten Sie auf eine realistische sowie kreative Farbgestaltung. Hilfreiche Dokumentation finden Sie in Ägypten[3] und im Zoo Hellabrunn[4].

Hinweis: Für den Erhalt der Bonuspunkte muss diese Aufgabe persönlich bei der Übungsleitung abgegeben werden.

Lösungsvorschlag 14.4

Für einen Lösungsvorschlag wenden Sie sich bitte an den Künstler bzw. Kamelexperten Ihres Vertrauens.

[3] `https://goo.gl/maps/ombVW2RrESt`

[4] `http://www.hellabrunn.de/`