Paperdiskussion SWS Seminar 2009

A Scheduling Framework for General-purpose Parallel Languages

(ICFP'08, September 22–24, 2008, Victoria, BC, Canada.)

Matthew Fluet
Toyota Technological Institute at Chicago
fluet@tti-c.org

Mike Rainey **John Reppy**
University of Chicago
{mrainey,jhr}@cs.uchicago.edu

Seminar geleitet von
Christoph Kirsch
Universität Salzburg
ck@cs.uni-salzburg.at

Präsentiert von
Alexander Baumgartner
Universität Salzburg
alexander.baumgartner@gmx.at

Schedulingframework

- Es geht **nicht** um einen bestimmten Schedulingalgorithmus
- Rahmenbedingungen rund um das Scheduling
  - Was sollte ich wissen wenn ich selbst einen Scheduler schreiben will. (Zum Beispiel: Wo ist das Scheduling eingehängt? Worum muss ich mich kümmern?...)
- Welche Möglichkeiten stehen einem (beliebigen) Scheduler zur Verfügung um das Scheduling durchzuführen
- Wie lässt sich ein Schedulingalgorithmus mit diesen Rahmenbedingungen und Möglichkeiten implementieren
- Ermöglicht einfaches Implementieren, Testen, Vergleichen,... von verschiedenen (bzw. neuen) Schedulingalgorithmen.

Gliederung des Papers

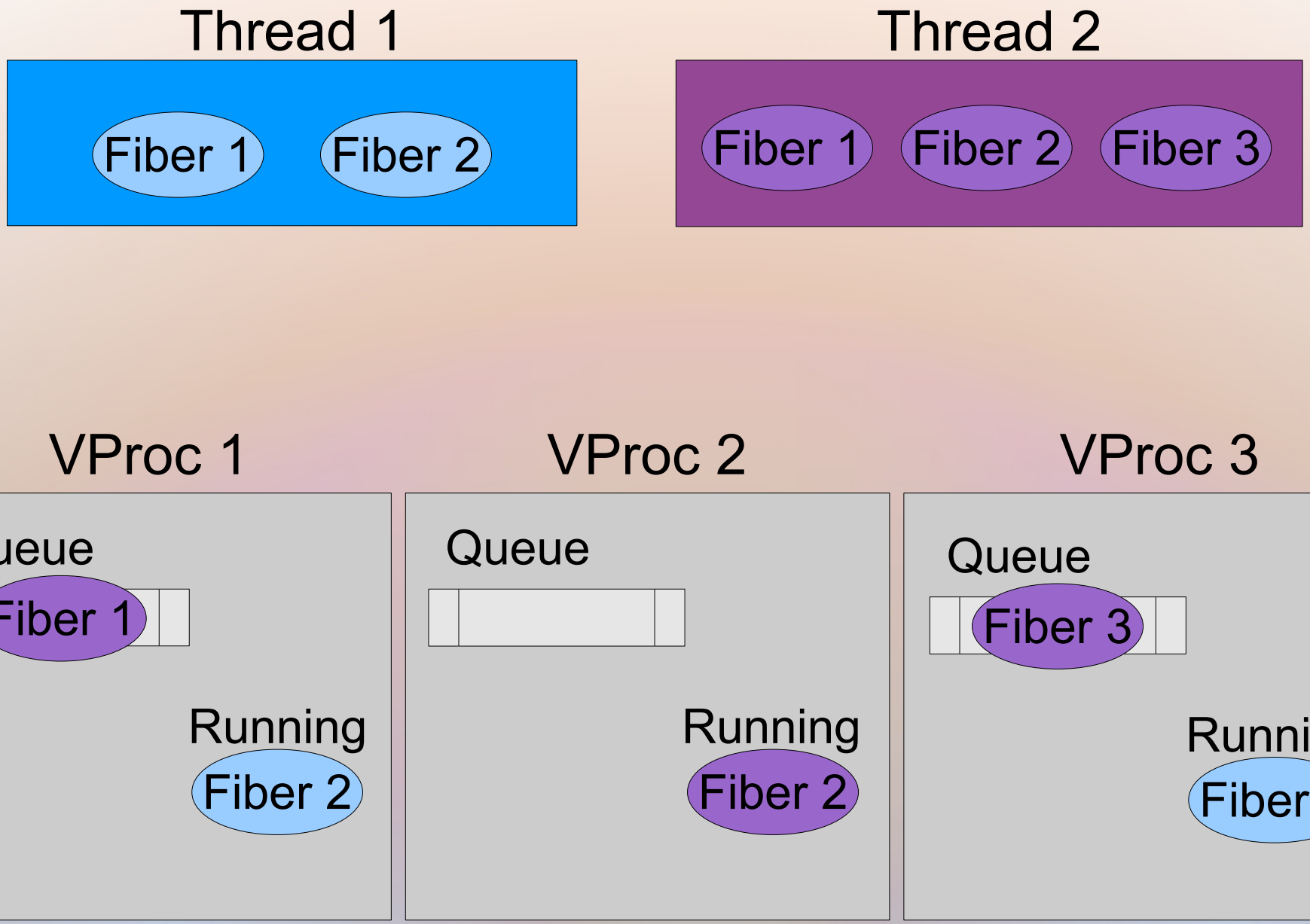
- Rahmenbedingungen
- Das Schedulingframework
- Implementierung verschiedener Schedulingalgorithmen
- Evaluierung der Skalierbarkeit von
  - Gang scheduling
  - Work stealing

Grundlegendes

- Das Schedulingframework wurde für Manticore entworfen und dort auch so implementiert
- Manticore ist eine parallele funktionale Programmiersprache
  http://manticore.cs.uchicago.edu/
  http://research.microsoft.com/en-us/um/people/crusso/ml2007/slides/manticore-ml07.pdf
- Der Code ist in SML (http://en.wikipedia.org/wiki/Standard_ML). Es wird viel mit Continuations gearbeitet.

Continuations

- Man kann sich eine Continuation als ein Aufsetzpunkt vorstellen.
  Es ist ein bestimmter Punkt in der Ausführung eines Programms, auf den man zu einem späteren Zeitpunkt wieder aufsetzen kann.
- **let cont** k arg = exp **in** body **end**
  - k — Der Name der Continuation
  - arg — Übergebene Parameter (Argumente)
  - exp — Der auszuführende Code beim Aufsetzen
  - body — Der auszuführende Code beim Erstellen der Continuation (body ist optional)
- Zum Aufsetzen auf eine existierende Continuation wird diese geschmissen.
- **throw** k ()
  - Es wird bei der Continuation k weitergemacht (keine Argumente)


Thread 1
Fiber 1
Fiber 2
Thread 2
Fiber 1
Fiber 2
Fiber 3
VProc 1
Queue
Fiber 1
Running
Fiber 2
VProc 2
Queue
Running
Fiber 2
VProc 3
Queue
Fiber 3
Running
Fiber 1

Fibers

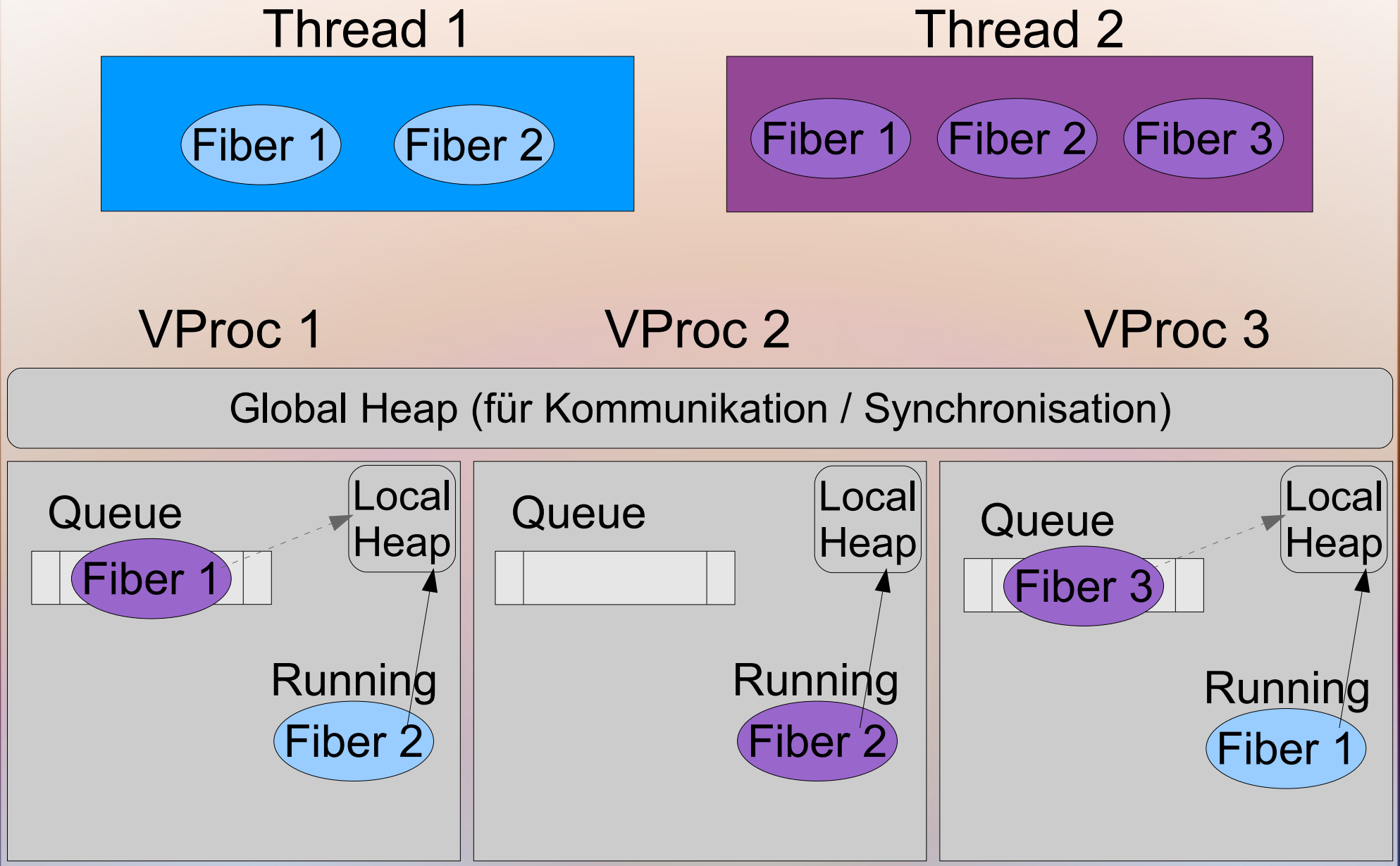
Fibers sind die Fasern eines Fadens

Diese Fasern werden zum Beispiel beim Sortieren eines parallelen Arrays oder beim Berechnen einer parallelen Variable implizit erzeugt.

Ein unterbrochener Fiber wird als Continuation auf der Queue abgelegt.

Fibers haben einen dynamischen lokalen Speicher.

Einige Befehle zum manipulieren dieses Speichers werden wir noch brauchen:
newFls(); getFls(); setFls(fls); getFromFls(fls, tag)


Thread 1
Fiber 1
Fiber 2
Thread 2
Fiber 1
Fiber 2
Fiber 3
VProc 1
VProc 2
VProc 3
Global Heap (für Kommunikation / Synchronisation)
Queue
Local Heap
Fiber 1
Running
Fiber 2
Queue
Local Heap
Running
Fiber 2
Queue
Local Heap
Fiber 3
Running
Fiber 1

Die Realisierung von Preemption

- Die Preemption ist mit dem garbage-collection test gekoppelt. Das ist praktisch, denn dann ist die Berechnung (computation) in einem sicheren Status.
- Es gibt einen einfachen pthread der in periodischen abständen SIGUSR2-Signale an alle vprocs (erweiterte pthreads) schickt.
- Jeder vproc hat einen Signalhandler, der bei einem SIGUSR2-Signal das heap-limit-Register auf 0 setzt.
- Beim nächsten heap-limit-Test wird GC ausgelöst. An dieser Stelle wird eine Continuation (Aufsetzpunkt) erstellt und an den Scheduler (des aktuellen vproc) weitergereicht.

Die Schedulingqueue

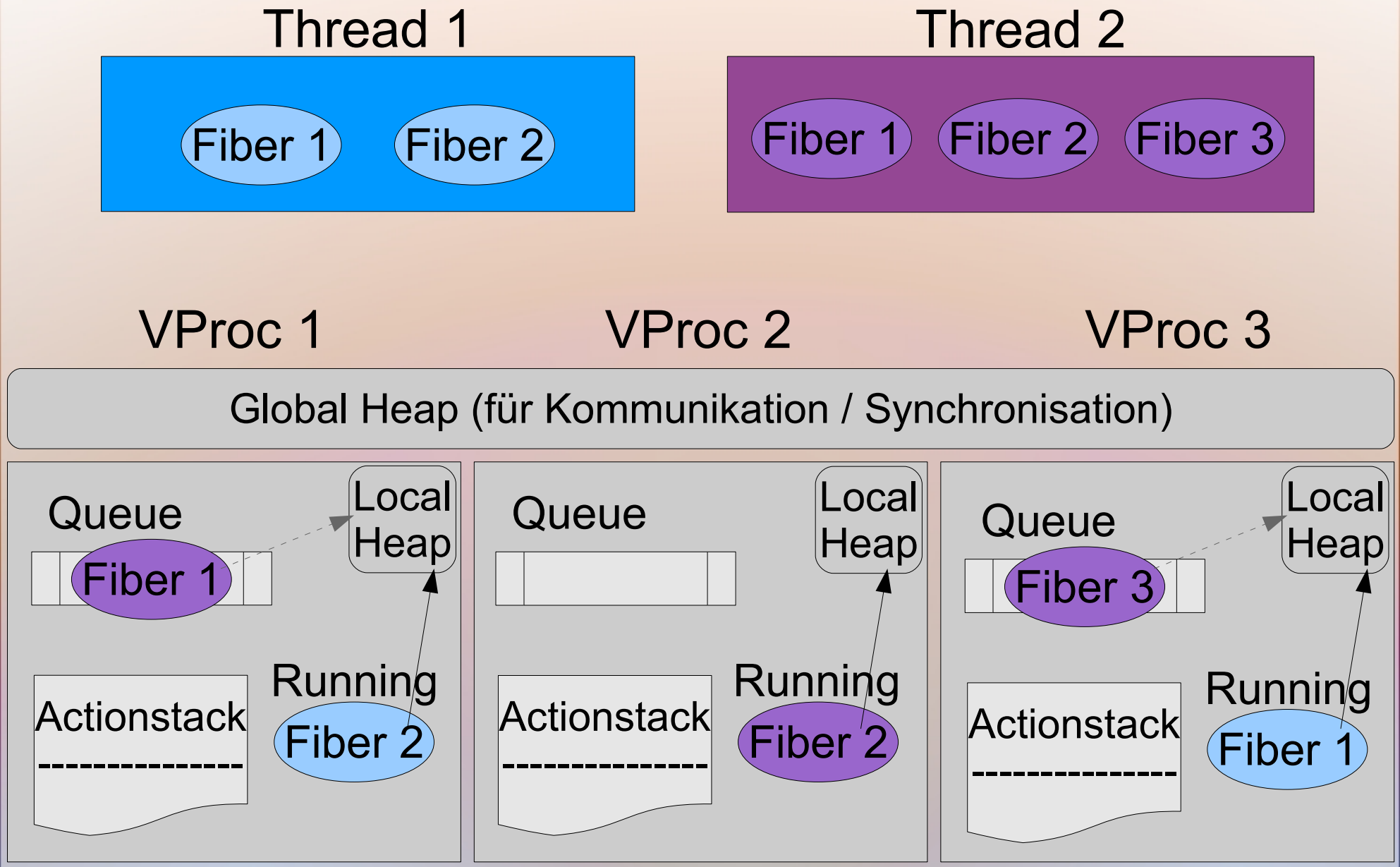
- Jeder vproc hat eine eigene Schedulingqueue.
- Sie wird in eine lokale und eine globale Queue aufgesplittet.
- Die lokale Queue hat keinen Synchronisierungsoverhead
- Die globale Queue wird durch einen Mutex gesichert
- Im Zuge der Preemtion wird als erstes die gesamte globale Queue in die lokale Queue verschoben

- Auch zum Arbeiten mit der Queue gibt es Befehle, die wir später noch brauchen werden:

  enq(fiber), deq(), enqOnVP(vproc, fiber)

Schedulingaktionen und -operationen

- Jeder vproc hat einen eigenen Aktionsstack um dort Schedulingaktionen (Schedulingalgorithmen) abzulegen.
- Eine Schedulingaktionen ist eine Continuation der ein Signal übergeben wird.
- Wir brauchen 2 Signale **STOP** und PREEMPT**.** Bei Zweitem brauchen wir die Continuation des aktuellen Fibers. Daher: **PREEMPT(cont)**
- Es gibt auch 2 Operationen die wir in unserem Schedulingcode verwenden können um den Aktionsstack zu verändern:
  - **run(actionCont, fiberCont)** Legt actionCont auf den Aktionsstack und schmeisst die Continuation fiberCont.
  - **forward(signal)** Nimmt die oberste Schedulingaktionen und schmeisst diese mit dem übergebenen Signal.


Thread 1
Fiber 1
Fiber 2
Thread 2
Fiber 1
Fiber 2
Fiber 3
VProc 1
VProc 2
VProc 3
Global Heap (für Kommunikation / Synchronisation)
Queue
Fiber 1
Local Heap
Actionstack
Running
Fiber 2
Queue
Local Heap
Actionstack
Running
Fiber 2
Queue
Fiber 3
Local Heap
Actionstack
Running
Fiber 1

Die Schedulingoperationen veranschaulicht

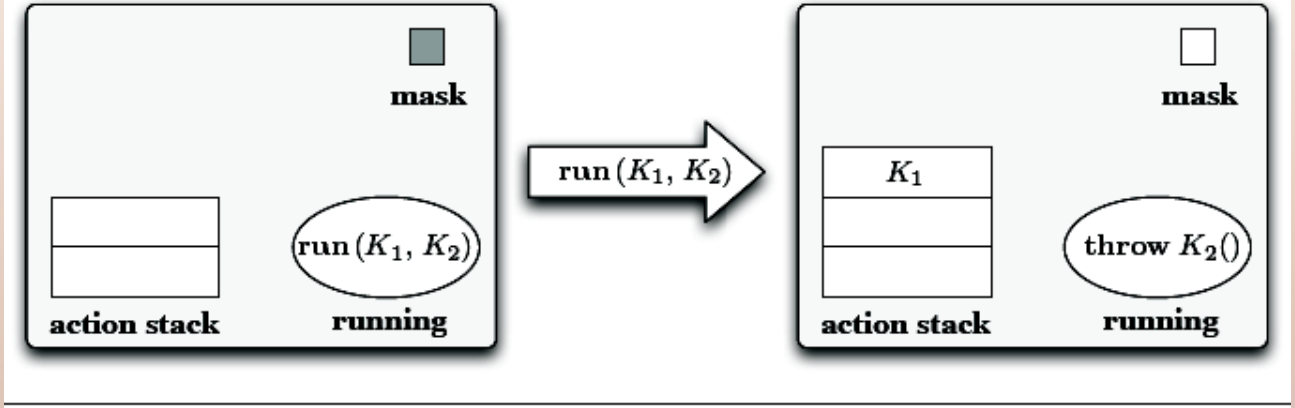


Figure 2. The effect of `run` (k_1, k_2) on a vproc

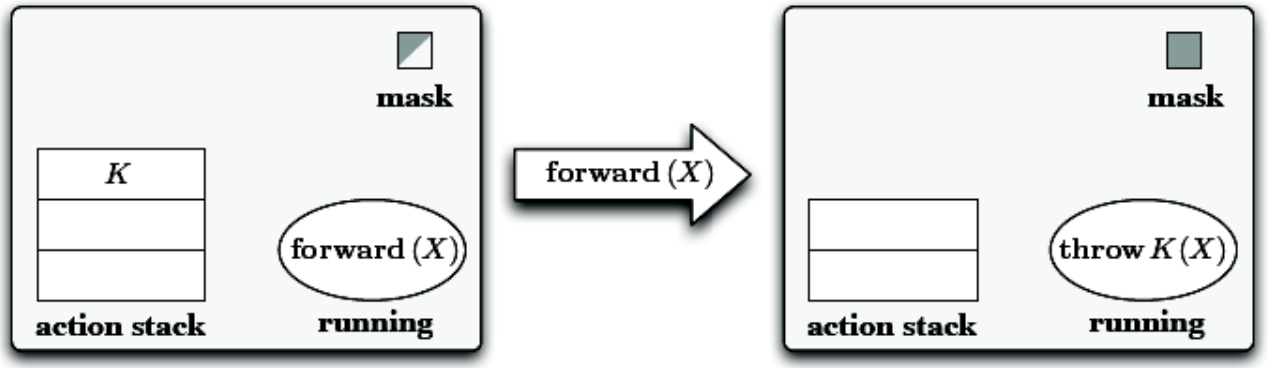


Figure 3. The effect of `forward` (sgn) on a vproc

Ein kurzer Überblick

- Auslöser:
  SIGUSR2 an vprocs-Signalhandler ⇒ heap-limit auf 0 ⇒ GC ⇒
- Preemption:
  Erstelle Continuation cont ⇒ forward(PREEMPT(cont)) ⇒
- Scheduler:
  Nimm Schedulingaktion (= Algorithmus = Continuation) und schmeiß diese Continuation mit dem übergebenen Signal ⇒
- Schedulingaktion hat 2 mögliche Signale:
  STOP ⇒ nächster Fiber aus der Schedulingqueue
  PREEMPT(cont) ⇒ stelle cont in die Queue ⇑
  ... und kümmere dich darum, dass wieder ein Aktion in den Stack ist!!!

Einfaches Beispiel: Round-Robin

```
cont    roundRobin (STOP) = dispatch()
    |   roundRobin (PREEMPT k) = let
        val fls = getFls()
        cont k' () = (
            setFls(fls);
            throw k () )
        in
            enq k';
            dispatch()
        end
cont dispatch () = run(roundRobin, deq())
```

Der Code ist in SML (http://en.wikipedia.org/wiki/Standard_ML).

User-Code (Höhere Funktionen)

- Mit unseren vorhandenen Mitteln können wir auch einfache Funktionen für den User zur Verfügung stellen. Zum Beispiel:
  - exit() Faden terminieren
    fun exit() = forward(STOP)
  - fiber(f) Baut aus einer Funktion eine Continuation (Fiber)
    fun fiber(f) = **let cont** K(X) = **let** () = f(X) **in** exit() **in** K
  - spawn(f) Baut einen neuen (User-)Thread
    fun spawn (f) = enq(fiber(**fn** () => (setFls(newFls()); f())))
  - yield(…
- Das Framework enthält einige solcher Funktionen, die natürlich auch beim Implementieren von Schedulingalgorithmen verwendet werden.

Voraussetzungen für Lastverteilung

- Das Laufzeitsystem bietet die Möglichkeit nach verfügbaren VProcs zu fragen.
- Es gibt nur ein **enqOnVP(vproc, fiber)** aber **kein** deqOnVP(vproc). Wir können also nur in eine fremde (globale) Queue hineinstellen aber nichts herausnehmen – das ist auch sinnvoll, wenn wir an die Architektur globale / lokale Queue denken.

Realisierung von Lastverteilung

- Wir reagieren erst, wenn ein VProc leerläuft.
  Der Scheduler erkennt, dass sich keine Continuation mehr in der Queue befindet.
- Jetzt fragen wir das Laufzeitsystem um andere zur Verfügung stehende VProcs.
- In diese anderen VProcs können wir Diebe (mit enqOnVp) einschleusen.
- Diese Diebe können (wieder mit enqOnVp) eine Continuation aus der fremden Queue in die „eigene“ Queue verschieben.

Parallele Operationen

- Parallele Variablen und Strukturen werden durch „Zukunftstypen“ (future) abgebildet.
- Die Laufzeitumgebung bietet eine Möglichkeit den FLS (fiber local storage) nach bestimmten Kriterien (Tags) zu durchsuchen.
- Daraus folgt: Im FLS kann nach solchen Futures gesucht werden. So wird implizite Parallelisierung ermöglicht.
- Natürlich muss sich der Scheduler auch darum kümmern.

Einen Thread abbrechen

- Das Nächste Problem, das sich stellt ist wenn der User einen Thread abbrechen will.
- Es wird folgende Datenstruktur definiert:
  datatype cancelable = CANCELABLE **of** {
  canceled : bool ref,
  inactive : bool ref,
  children : cancelable list ref,
  parent : cancelable option
  }
- Jedes Kind ist selbst wieder ein cancelable
- Damit wird es möglich alle Kinder zu canceln und das canceled-Flag wird gesetzt. Kommt ein Kind nun an die Reihe, dann werden wieder alle Kinder gecancelt... Baumförmige Fortsetzung.

Schlussworte zum Paper

- Es sind einige interessante Ideen enthalten.

  Das Implementieren eines eigenen Algorithmus ist sehr einfach (zumindest sieht es so aus).
  Dadurch wird jede denkbare Schedulingstrategie ermöglicht.
- Es sind jede Menge Details und Beispiele vorhanden.
- Zum Schluss des Papers wurden noch die Schedulingalgorithmen „Gang scheduling“ und „Work stealing“ unter verschiedenen Bedingungen in Manticore auf Skalierbarkeit getestet.
  (Der gesamte Code ist auch im Paper enthalten und beschrieben)
- Das Paper ist sehr interessant aber nicht einfach zu verstehen.

ENDE

DANKE FÜR EURE AUFMERKSAMKEIT