Large scale separation through monads: The Oregon Separation Kernel (previously Pauli separation kernel)

Peter White
OGI School of Science and Engineering
Oregon Health & Science University
muritza@attbi.com
Overview

- Process separation built from kernel thread separation
  - Kernel threads are under control of the kernel designer, process code is not
- Kernel thread separation built mostly from types, and a little bit from theorem proving
  - Kernel thread must be a rich enough structure to support programming the API calls
Process separation built from kernel thread separation

- Each user process (user half, or u/2) is an abstraction created by its corresponding kernel thread, called the system half (s/2)
- The kernel core is the initial thread, controlling the creation of other threads in the system
Outline

• Construction of the kernel thread
  – Rich enough structure in which to implement the API
    • Function calls
    • Mutable state
    • Exceptions
    • Interleaved execution

• Construction of the braid
  – Multi threading environment
    • Kernel (braid) calls
    • Separation property between properly constructed threads
Outline, step 1

- Construction of the kernel thread
  - Rich enough structure in which to implement the API
    - Function calls
    - Mutable state
    - Exceptions
    - Interleaved execution

- Construction of the braid
  - Multi threading environment
    - Kernel calls
    - Separation property between properly constructed threads
### Osker Criteria for a thread structure

- **Separation**: How much of thread separation is captured in the types?
  - Process code: Can the process cause separation to fail without the assistance of the kernel calls?
  - Kernel code: Does privileged kernel code violate the separation property?
- **Adequacy**: How easy is it to program API calls in the structure?
- **Swept under the rug**: How much of thread separation depends on advanced features of the run time system?
- **Speed**: How many thread schedules can be performed in one second?
- **Features**: Do we have mutable state, exceptions, interleaving, and kernel calls?

<table>
<thead>
<tr>
<th>Structure</th>
<th>Separation / process</th>
<th>Separation / kernel</th>
<th>Swept Under Rug</th>
<th>Performance</th>
<th>Mutable State</th>
<th>Exceptions</th>
<th>Interleaving</th>
<th>Kernel Calls / Exec</th>
</tr>
</thead>
</table>
Three sample threads

• Policy
  – A can communicate to (interfere with) B
  – B can communicate to (interfere with) C
  – A cannot directly interfere with C
    • Any interference of A with C must be mediated by B
    • If the result of running C is affected by the prior execution of A, then there must have been an execution of B between the executions of A and C

thread A (red)

\[\begin{align*}
b_1 & \leftarrow \text{check "A->B"} \\
d_1 & \leftarrow \text{createQ "A->B" send} \\
& \quad \text{when (bad \(d_1\))} \\
& \quad \quad \text{(error "err.1")} \\
& \quad \text{let } y = f\ x_1\ d_1 \\
& \quad \text{sendQ } d_1\ y \\
& \quad \text{cOk } \leftarrow \text{closeQ } d_1 \\
& \quad \text{uOk } \leftarrow \text{unlinkQ } d_1 \\
& \quad \text{return } ()
\end{align*}\]

thread B (crypto)

\[\begin{align*}
b_2 & \leftarrow \text{checkQ "A->B"} \\
d_1 & \leftarrow \text{openQ "A->B" rcv} \\
& \quad \text{x } \leftarrow \text{receive } d_1 \\
& \quad \text{m } d_2 \leftarrow \text{create "B->C" send} \\
& \quad \quad \text{sendQ } d_2\ (g\ x_2\ x) \\
& \quad \text{cOk}_1 \leftarrow \text{closeQ } d_1 \\
& \quad \text{cOk}_2 \leftarrow \text{closeQ } d_2 \\
& \quad \text{uOk}_1 \leftarrow \text{unlinkQ } d_1 \\
& \quad \text{uOk}_2 \leftarrow \text{unlinkQ } d_2 \\
& \quad \text{return } ()
\end{align*}\]

thread C (black)

\[\begin{align*}
b_1 & \leftarrow \text{checkQ "B->C"} \\
d_2 & \leftarrow \text{openQ "B->C" rcv} \\
& \quad \text{x } \leftarrow \text{receiveQ } d_2 \\
& \quad \text{cOk } \leftarrow \text{closeQ } d_2 \\
& \quad \text{return } ()
\end{align*}\]
Local and non local processing

• Goals:
  – Make local processing provably local via types
  – Establish (mostly via types) that the only non-local process is via the thread primitives (check, create, send, …) provided
  – Establish (mostly via types) that the thread primitives are used in accordance with policy
Dividing a thread into steps

As a sequence of steps

\[ b_1 \leftarrow \text{checkQ } "A\rightarrow B" \]
\[ \text{md}_1 \leftarrow \text{createQ } "A\rightarrow B" \text{ send} \]
\[ \text{sendQ } d_1 \left( f \, x_1 \ "A\rightarrow B" \) \]
\[ \text{closeQ } d_1 \]

As a sequence of functions of state

\[ \text{checkQ } "A\rightarrow B" : s \rightarrow (s, \text{Bool}) \]
\[ \text{Form } \lambda s \rightarrow (s, \text{Bool}) \]
\[ \text{openQ } "A\rightarrow B" \text{ flags} : s \rightarrow (s1 \text{ Desc}) \]
\[ \text{Form } \lambda s \rightarrow (s, \text{Descriptor}) \]
\[ \text{sendQ } d_1 \left( f \, x_1 \ "A\rightarrow B" \) : s \rightarrow (s,()) \]
\[ \text{Form } \lambda s \rightarrow (s,()) \]
\[ \text{closeQ } d_1 : s \rightarrow (s,()) \]
\[ \text{Form } \lambda s \rightarrow (s,()) \]

- Goal: to be able to interleave the steps of one thread with the steps of other threads
Putting the steps back together

As a sequence of steps

\[ b_1 \leftarrow \text{checkQ "A->B"} \]

\[ \text{‘bind’} \]

\[ \text{md}_1 \leftarrow \text{createQ “A->B” send} \]

\[ \text{‘bind’} \]

\[ \text{sendQ d}_1 (f x_1 \text{“A->B”}) \]

\[ \text{‘bind’} \]

\[ \text{closeQ d}_1 \]

As a sequence of functions of state

\[ \text{checkQ “A->B” :: s \rightarrow (s, \text{Bool})} \]

\[ \text{Form } \lambda s \rightarrow (s, \text{Bool}) \]

\[ \text{openQ “A->B” flags :: s \rightarrow (s1 Desc)} \]

\[ \text{Form } \lambda s \rightarrow (s, \text{Descriptor}) \]

\[ \text{sendQ d}_1 (f x_1 \text{“A->B”}) :: s \rightarrow (s,()) \]

\[ \text{Form } \lambda s \rightarrow (s,()) \]

\[ \text{closeQ d}_1 :: s \rightarrow (s,()) \]

\[ \text{Form } \lambda s \rightarrow (s,()) \]

- Goal: to be able to interleave the steps of one thread with the steps of other threads
State monad hides the state parameter

\[ \text{State monad} = S(\lambda s \rightarrow (s, a)) \]
The local state of the thread evolves with each bind / compose

Separation depends more on the bind combinator than on the operations bound
Thread monad summary

- **Thread monad has type**
  - data Thread s a = Thread (s -> (s, a))
  - bind: Plumbs state from one operation into the next operation.

- **Thread monad captures mutable state**

- **Functions in a thread are safe**
  - State change in one thread does not affect another thread, without help from an executive

- **Thread monad does not offer interleaving or exceptions**
  - Each thread is run to completion

- **No executive or kernel calls have been specified for threads**

- **Separation within a thread is good, but the executive could plumb the state of one thread through another thread**

<table>
<thead>
<tr>
<th>Structure</th>
<th>Separation/process</th>
<th>Separation/kernel</th>
<th>Adequacy</th>
<th>Swept Under Rug</th>
<th>Performance</th>
<th>Mutable State</th>
<th>Exceptions</th>
<th>Interleaving</th>
<th>Kernel Calls/Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread monad</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Braiding the threads

- Description of Braid 0 type
  - data Braid s a = Braid (s -> (s, a))
  - The state (s) is specialized to the state depicted on the right
  - bind: Selects thread to run, runs it, updates relevant state information

- Each thread has its own mutable state

- Braid 0 monad does not offer interleaving or exceptions
  - Each thread is run to completion

- Can implement the kernel calls (openQ, sendQ, …)
  - The executive is the braid itself

- Separation
  - Process: Local processing separated
  - Kernel calls: No separation property offered
    • Program of type Braid s a = Braid(s -> (s, a)) can make arbitrary updates to the state s.

---

<table>
<thead>
<tr>
<th>local program</th>
<th>tid₁ prog(1,1)</th>
<th>tid₂ prog(2,1)</th>
<th>tid₃ prog(3,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>local state</td>
<td>tid₁ ls(1,1)</td>
<td>tid₂ ls(2,1)</td>
<td>tid₃ ls(3,1)</td>
</tr>
<tr>
<td>queues</td>
<td>A-&gt;B [y]</td>
<td>B-&gt;C [w]</td>
<td></td>
</tr>
<tr>
<td>threadstate</td>
<td>tid₁ running</td>
<td>tid₂ ready</td>
<td>tid₃ ready</td>
</tr>
<tr>
<td>current</td>
<td>tid₁</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Structure</th>
<th>Separation / process</th>
<th>Separation / kernel</th>
<th>Adequacy</th>
<th>Swept Under Rug</th>
<th>Performance</th>
<th>Mutable State</th>
<th>Exceptions</th>
<th>Interleaving</th>
<th>Kernel Calls / Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread monad</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Thread Braid 0 / unlifted</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
**State evolution in the braid 0 monad**

```
run tid₁

sendQ d₁ y
```

- Local program has type Braid s ()
  - Braid is a recursive data type
Lifting an isolated thread into the braid results in a computation that runs as if it were isolated within the context of the braid.
Transitive interference

{-P: {< Braid internal separation through lifting property >-}
-- The separation through lifting property, for transitive null policy
assert SeparateLiftTransitive =
  All sched :: P.Schedule.
  All filt :: ThreadFilter.
  All bst :: BraidSt.
  All tinit :: [TH.ThreadInit (Thread ())].
  { filterObservationsTransitive
    sched filt (observations sched (lifts bst tinit))
  } ===
  { observations (filterScheduleTransitive sched filt)
    (lifts bst tinit) }

\[ \text{dom}_A \]
\[ \text{dom}_B \]
\[ \text{dom}_C \]
{-P: {<- The intransitive separation through lifting property.>-
assert SeparateLiftIntransitive =
    All sched :: P.Schedule.
    All policy :: P.Policy.
    All target :: TID.ThreadId.
    All bst :: BraidSt.
    All tinit :: [TH.ThreadInit (Thread ())].
    { filterObservationsIntransitive
        policy target (observations sched (lifts bst tinit))
    } ===
    { observations (filterScheduleIntransitive sched policy target)
        (lifts bst tinit)
    }
}
Interference cause

assert InterferenceCause =

All bst :: BraidSt.
All tid1 :: TID.ThreadId.
All th1 :: TH.ThreadInit SingleThread.
All tid2 :: TID.ThreadId.
All th2 :: TH.ThreadInit SingleThread.

Interference1 tid1 th1 tid2 th2 ==> 

( / ( { runTid tid1 bst >> getState tid2 } == { getState tid2 } 

• If there is interference, it is caused by another thread (tid1) affecting those components of the state that are relevant to the execution of tid2
  – Contents of the queues
  – tid1 local state (ruled out for lifted threads)
  – tid1 program (ruled out for lifted threads)
  – currently running tid (ruled out for lifted threads)
  – tid1 thread state (ruled out for lifted threads)
Braid 0 summary

• Lifting of threads yields separation
  – Lifted threads enjoy separation property by virtue of:
    • Their types (mostly)
    • Correctness of the lift operation (a little theorem proving)

<table>
<thead>
<tr>
<th>Structure</th>
<th>Separation / process</th>
<th>Separation / kernel</th>
<th>Adequacy</th>
<th>Swept Under Rug</th>
<th>Performance</th>
<th>Mutable State</th>
<th>Exceptions</th>
<th>Interleaving</th>
<th>Kernel Calls / Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread monad</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Thread Braid 0 / unlifted</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>local / lifted</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>
Braid 1 (exceptions) summary

- Description of Braid 1 type
  - data Braid s a = Braid (s -> (s, E a))
  - bind:
    - Plumbs state from one operation into the next
    - Propagates exceptions through the bind
- Functions executed within a thread are safe
- Still no interleaving
  - Each thread is run to completion
- Kernel calls
  - Can add intra thread (throw) and inter thread (throwTo) exceptions
  - Kernel calls still assured by analysis
    - Analysis assisted by type safety
- Braid state
  - Add an exception handler program per thread

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread monad</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Thread Braid 0 / unlifted</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>local / lifted</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Braid 1</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>


• Exception catcher is associated with each thread id
  – Catcher and program have type Braid1 s ()
    • Braid1 is a recursive type
  – The catcher is changed by the `catch` kernel call

<table>
<thead>
<tr>
<th>local catcher</th>
<th>tid₁ (catch(1,1))</th>
<th>tid₂</th>
<th>tid₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>local program</td>
<td>tid₁ (prog(1,1))</td>
<td>tid₂ ,1)</td>
<td>tid₃ ,1)</td>
</tr>
<tr>
<td>local state</td>
<td>tid₁ (ls(1,1))</td>
<td>tid₂ ls(2,1)</td>
<td>tid₃ ls(3,1)</td>
</tr>
<tr>
<td>queues</td>
<td>A-&gt;B [y]</td>
<td>B-&gt;C [w]</td>
<td></td>
</tr>
<tr>
<td>threadstate</td>
<td>tid₁ running</td>
<td>tid₂ ready</td>
<td>tid₃ ready</td>
</tr>
<tr>
<td>current</td>
<td>tid₁</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Separation in braid 1

• The separation properties in braid 1 are identical to those in braid 0
Braid 2 (interleaving)

- **Description of braid 2 type**
  - data RSEVal s a
  - = Continue s (E a)
  - | Pause s (RSE s a)
  - Data RSE s a = RSE (s -> RSEVal s a)
  - bind:
    - Plumbs state from one operation into the next
    - Propagates exceptions through the bind
    - Permits continuation of the computation or a pause in the computation
- **Functions executed within a thread are safe**
- **Braid**
  - Braid2 is specialized to the internal state shown
  - Each catcher and thread program have type Braid2 ()
    - Braid2 is a recursive type
Braid2 (interleaving) summary

- Braid 2 is an adequate environment to program POSIX calls
- About 140000 thread switches per second
  - Little effort has been put into optimization
- We still have the correctness of the kernel calls to worry about
- Braid 2 is purely functional, no reliance on the Haskell IO threads
  - “Under the rug”
    - Functional evaluation via thunks
    - Garbage collection

<table>
<thead>
<tr>
<th>Structure</th>
<th>Separation / process</th>
<th>Separation / kernel</th>
<th>Adequacy</th>
<th>Swept Under Rug</th>
<th>Performance</th>
<th>Mutable State</th>
<th>Exceptions</th>
<th>Interleaving</th>
<th>Kernel Calls / Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread monad</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Thread Braid 0 / unlifted</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>local / lifted</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Braid 1</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Braid 2</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Separation in braid 2

- The separation properties in braid 2 are identical to those in braid 0
The end game: Process separation

- Each domain is a braid
  - Supporting process separation
- Domains get their own “separation through lifting” property
- Within the braid, processes are separated via their underlying kernel threads
  - Kernel thread performs local services for a user process
  - The user program can be interpreted, simulating running the program on the hardware
  - Interpreter is under control of the system half, and can be interrupted at any time
  - Multi threading is cooperative, but only at the level of kernel threads
Taming kernel calls

- Kernel calls are already quite tame
- Underlying functionality in RSE monad is 149 LOC
- Braiding the threads requires an additional 664 LOC
- Braid has a small list of data abstractions and methods upon which the separation properties depend

<table>
<thead>
<tr>
<th>Channels</th>
<th>Braid</th>
<th>Resumption / State / Exception Monad</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVar</td>
<td>newEmptyMVar</td>
<td>149 LOC</td>
</tr>
<tr>
<td>Exception</td>
<td>newMVar</td>
<td></td>
</tr>
<tr>
<td>ThreadId</td>
<td>deleteMVar</td>
<td></td>
</tr>
<tr>
<td>Thread</td>
<td>takeMVar</td>
<td></td>
</tr>
<tr>
<td>modifyMVar</td>
<td>putMVar</td>
<td></td>
</tr>
<tr>
<td>withMVar</td>
<td>swapMVar</td>
<td></td>
</tr>
<tr>
<td>weave</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Abstractions</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>readMVar</td>
<td>yield</td>
</tr>
<tr>
<td>fork</td>
<td>fork</td>
</tr>
<tr>
<td>killThread</td>
<td>threadDelay</td>
</tr>
<tr>
<td>myThreadId</td>
<td>myThreadId</td>
</tr>
<tr>
<td>throw</td>
<td>throwTo</td>
</tr>
<tr>
<td>throwTo</td>
<td>catch</td>
</tr>
</tbody>
</table>
Structured and restricted kernel calls

getTidState tid₁
MVar access policy is enforced, selecting only those MVars to which tid₁ has access

updateTidState tid₁

catch h :: TidState -> TidState

catch (1,2) = h
Structured kernel calls

- Another way to look at it: Refactoring
  \[ f \text{ tid } a \ b \ c = f' a \ b \ c ( \text{getTidState tid} ) \]
Structured / restricted kernel calls summary

- `getTidState` and `updateTidState` must correctly select and update state components.
- Kernel call, by their type (TidState -> TidState) can only affect their own state.
- Reduce trusted LOC count to 400.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Separation / process</th>
<th>Separation / kernel</th>
<th>Adequacy</th>
<th>Swept Under Rug</th>
<th>Performance</th>
<th>Mutable State</th>
<th>Exceptions</th>
<th>Interleaving</th>
<th>Kernel Calls / Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread monad</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Thread Braid 0 / unlifted</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>local / lifted</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Braid 1</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Braid 2</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Structured Kernel Calls</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
Osker to do

• Interrupts
  – User programs run under interpreter are interruptible
  – Kernel threads currently use cooperative multitasking
    • Looking at ways to make kernel threads interruptible, to support device drivers

• Features
  – Job control, pipes, …
Osker summary

• Osker is currently 25000 LOC
  – 400 trusted for thread separation property
• Have achieved the “mostly by types, a little by theorem proving” goal for the architecture
• The thread switching performance is excellent (140000 per second)
• Very little is under the rug
The bottom line

- The framework of Osker supports separation in large scale software projects
  - Complete separation (MILS)
  - Intransitive interference (MLS and other policies)