Large scale separation through monads: The Oregon Separation Kernel

(previously Pauli separation kernel)

Peter White

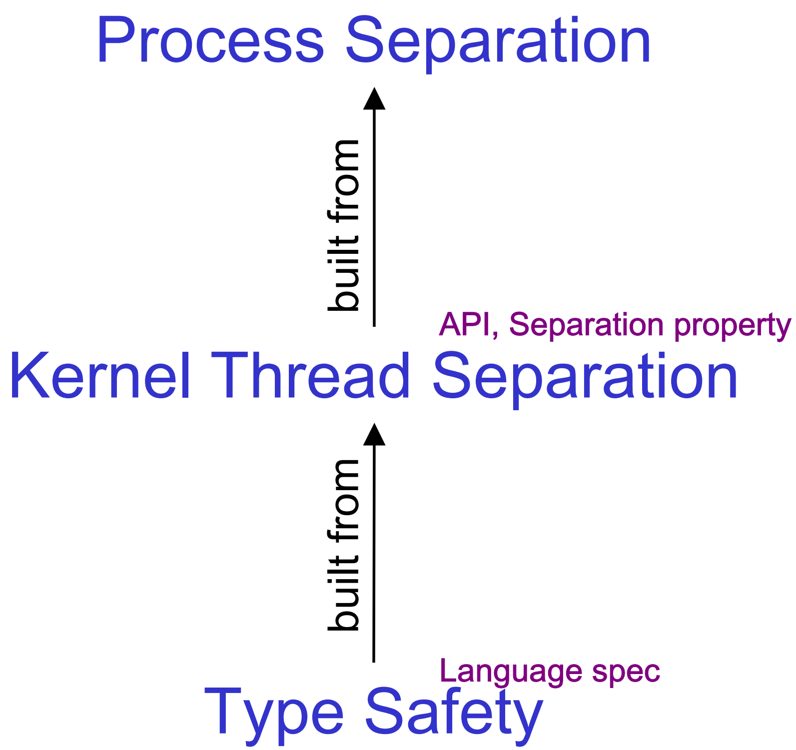
OGI School of Science and Engineering

Oregon Health & Science University

muritza@attbi.com

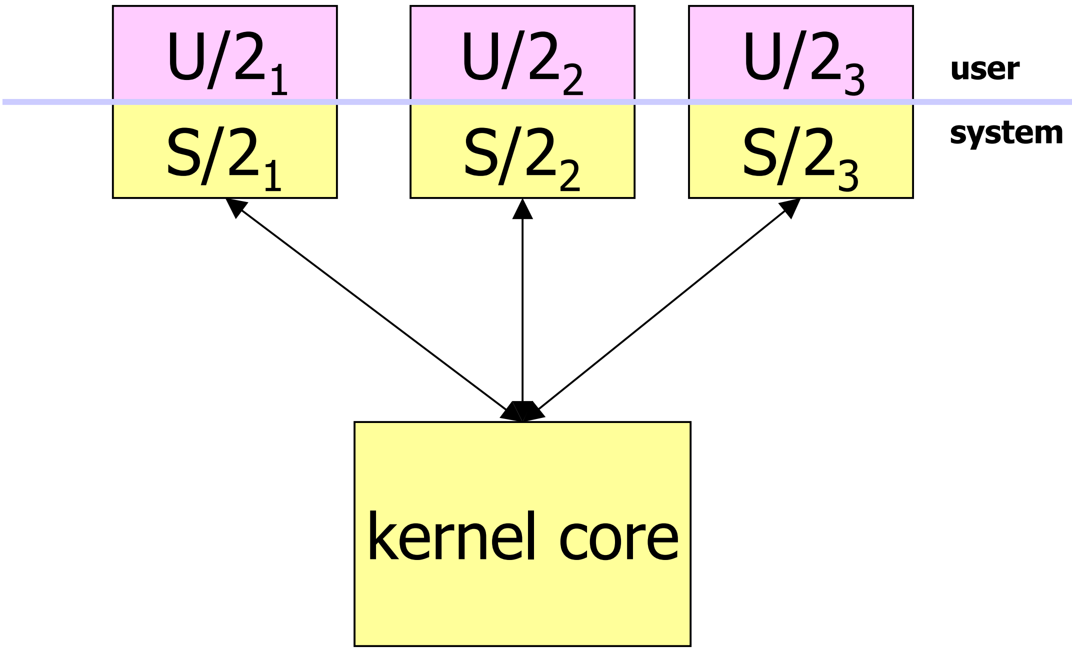
Overview

- 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?

Structure	Separation / process	Separation / kernel	Sufficiency	Swept Under Rug	Performance	Mutable State	Exceptions	Interleaving	Kernel Calls / Exec

Three sample threads

x_1 → thread A; x_2 → thread B

thread A (red)

```
b1 <- check "A->B"
d1 <-createQ "A->B" send  ①
when (bad d1)
     (error "err.1")
let y = f x1 d1
sendQ d1 y ②
cOk <- closeQ d1
uOk <- unlinkQ ‹ ③
return ()
```

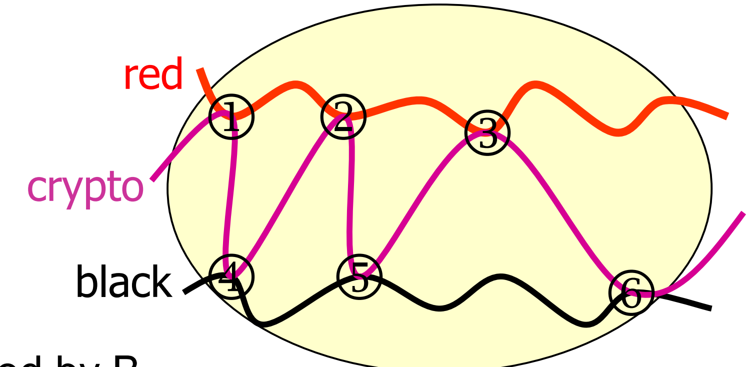
thread B (crypto)

```
b2 <- checkQ "A->B"
d1 <- openQ "A->B" rcv
x <- receive d1
m d2 <- create "B->C" send
sendQ d2(g x2 x)
cOk1 <- closeQ d1
cOk2 <- closeQ d2
uOk1 <- unlinkQ d1
uOk2 <- unlinkQ d2
return ()
```

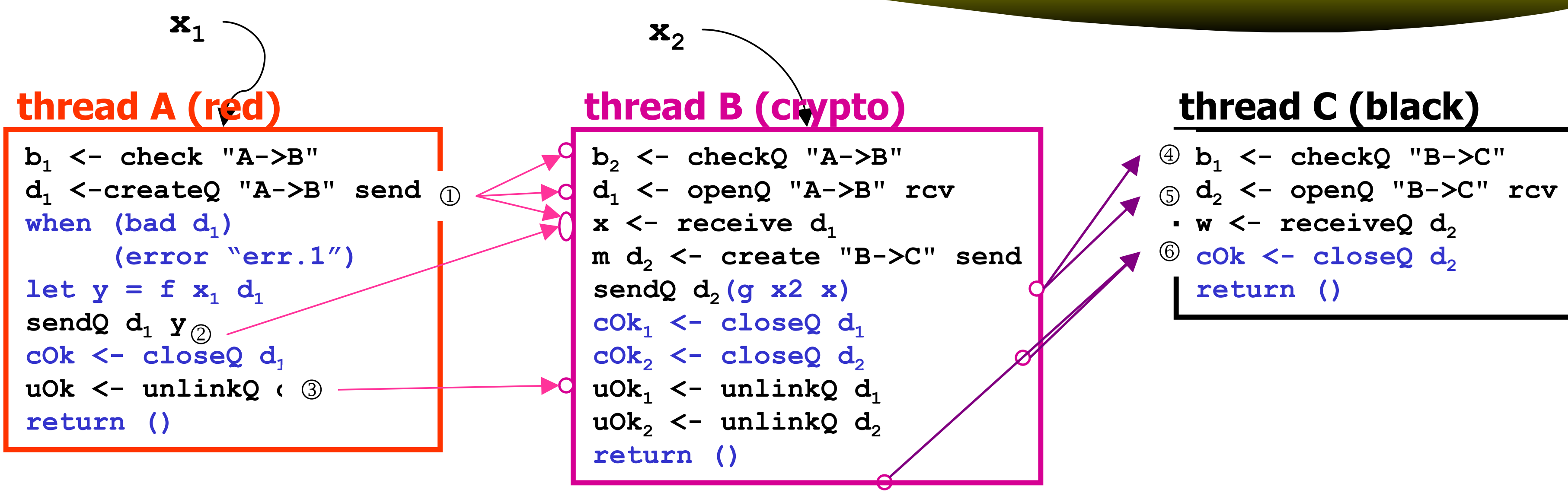
thread C (black)

```
④ b1 <- checkQ "B->C"
⑤ d2 <- openQ "B->C" rcv
⑥ w <- receiveQ d2
  cOk <- closeQ d2
  return ()
```

- 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



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

```
b1 <- checkQ "A->B"

md1 <- createQ "A->B" send

sendQ d1 (f x1 "A->B" )

closeQ d1
```

As a sequence of steps

```
checkQ "A->B" :: s -> (s, Bool)
Form λs ->(s, Bool)

openQ "A->B" flags ::s -> (s1 Desc)
Form λs ->(s, Descriptor)

sendQ d1 (f x1 "A->B" ):: s -> (s,())
Form λs ->(s, ())

closeQ d1 :: s -> (s, ())
Form λs ->(s, ())
```

As a sequence of functions of state

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

Putting the steps back together

```
b₁ <- checkQ "A->B"
    'bind'
md₁ <- createQ "A->B"  send
    'bind'
sendQ d₁ (f x₁ "A->B" )
    'bind'
closeQ d₁
```

```
checkQ "A->B" :: s -> (s, Bool)
Form λs ->(s, Bool)

openQ "A->B" flags ::s -> (s1 Desc)
Form λs ->(s, Descriptor)

sendQ d₁ (f x₁ "A->B" ):: s -> (s,())
Form λs ->(s, ())

closeQ d₁ :: s -> (s, ())
Form λs ->(s, ())
```

As a sequence of steps | As a sequence of functions of state

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

State monad hides the state parameter

b_1 <- checkQ "A->B"

'bind'

md_1 <- createQ "A->B" send

'bind'

sendQ d_1 (f x_1 "A->B")

'bind'

closeQ d_1

s_0

(s_1, b_1)= checkQ "A->B" s_0

'compose'

(s_2, d_2)= openQ "A->B" flags s_1

'compose'

(s_3,()) = sendQ d_1 (f x_1 "A->B") s_2

'compose'

(s_4, ()) = closeQ d_1 s_3

checkQ "A->B" :: s -> (s, Bool)
Form λs ->(s, Bool)

openQ "A->B" flags ::s -> (s1 Desc)
Form λs ->(s, Descriptor)

sendQ d_1 (f x_1 "A->B"):: s -> (s,())
Form λs ->(s, ())

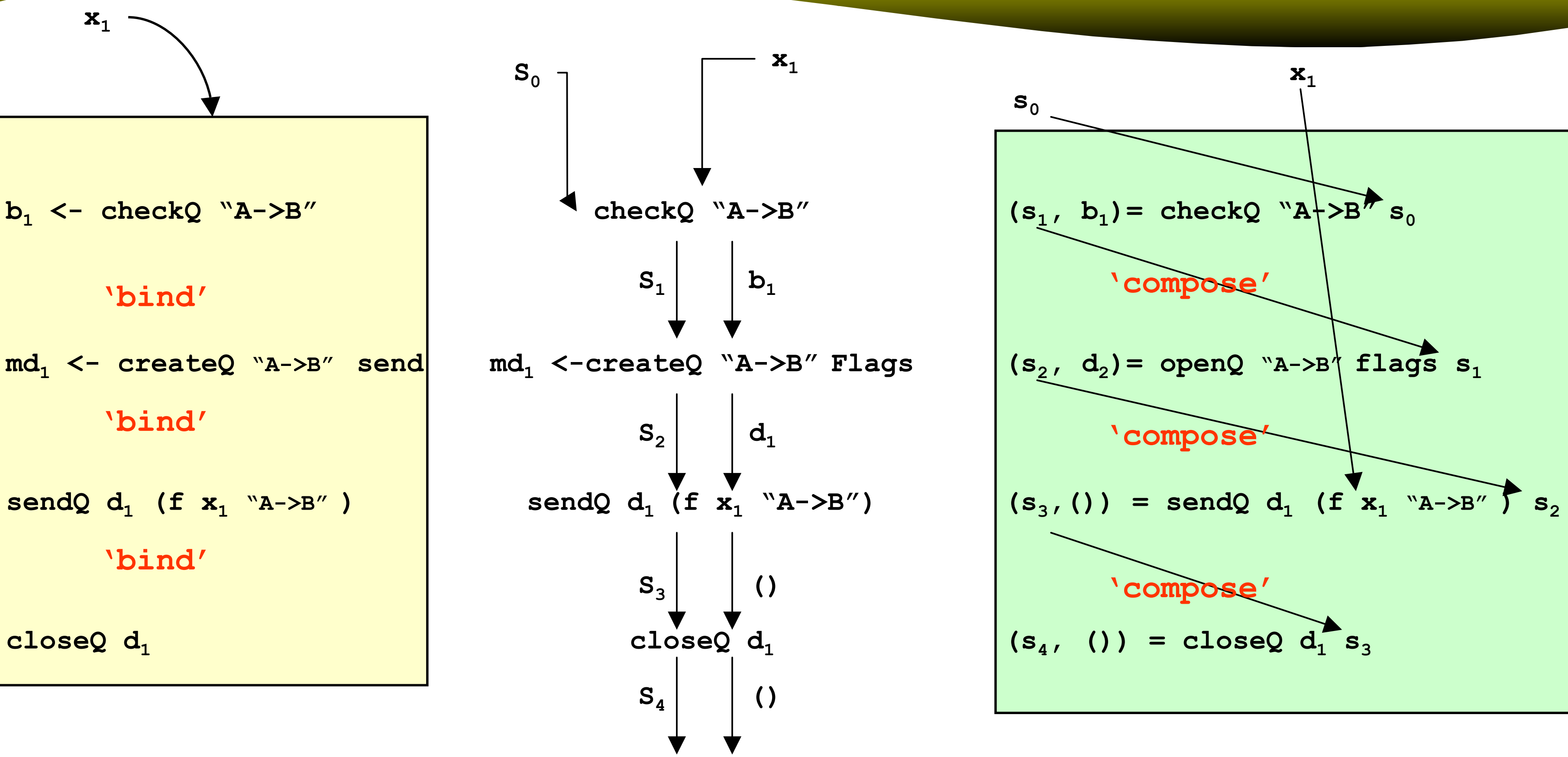
closeQ d_1 :: s -> (s, ())
Form λs ->(s, ())

thread A :: Name->x-> thread ()
thread A u_1 x_1 =
do { b_1 <- check "A-> B"
; d_1 <- open "A->B" flags
; send d_1 (f x_1 n_1)
; close d_1
}

thread A :: s-> Name->x-> (s,())
thread A s_0 n_1 x_1 =
let (s_1, b_1) = check n_1 s_0
(s_2, d_1) = open n_1 flags s_1
(s_3,()) = send d_1 (f x_1 n_1) s_2
(s_4,()) = close d_1 s_3
in (s_4,())

General form of the state monad
Form λs ->(s, a)
State Monad = S(λs ->(s, a))

Thread State monad in pictures



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

Structure	Separation / process	Separation / kernel	Adequacy	Swept Under Rug	Performance	Mutable State	Exceptions	Interleaving	Kernel Calls / Exec
Thread monad	3	1	1	5	5	Y	N	N	N

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.

local program	tid_1 prog(1,1)	tid_2 prog(2,1)	tid_3 prog(3,1)
local state	tid_1 ls(1,1)	tid_2 ls(2,1)	tid_3 ls(3,1)
queues	A->B [y]	B->C [w]	
threadstate	tid_1 running	tid_2 ready	tid_3 ready
current	tid_1		

Structure	Separation / process	Separation / kernel	Adequacy	Swept Under Rug	Performance	Mutable State	Exceptions	Interleaving	Kernel Calls / Exec
Thread monad	3	1	1	5	5	Y	N	N	N
Thread Braid 0 / unlifted	3	1	2	5	5	Y	N	N	Y

State evolution in the braid 0 monad

run tid_1

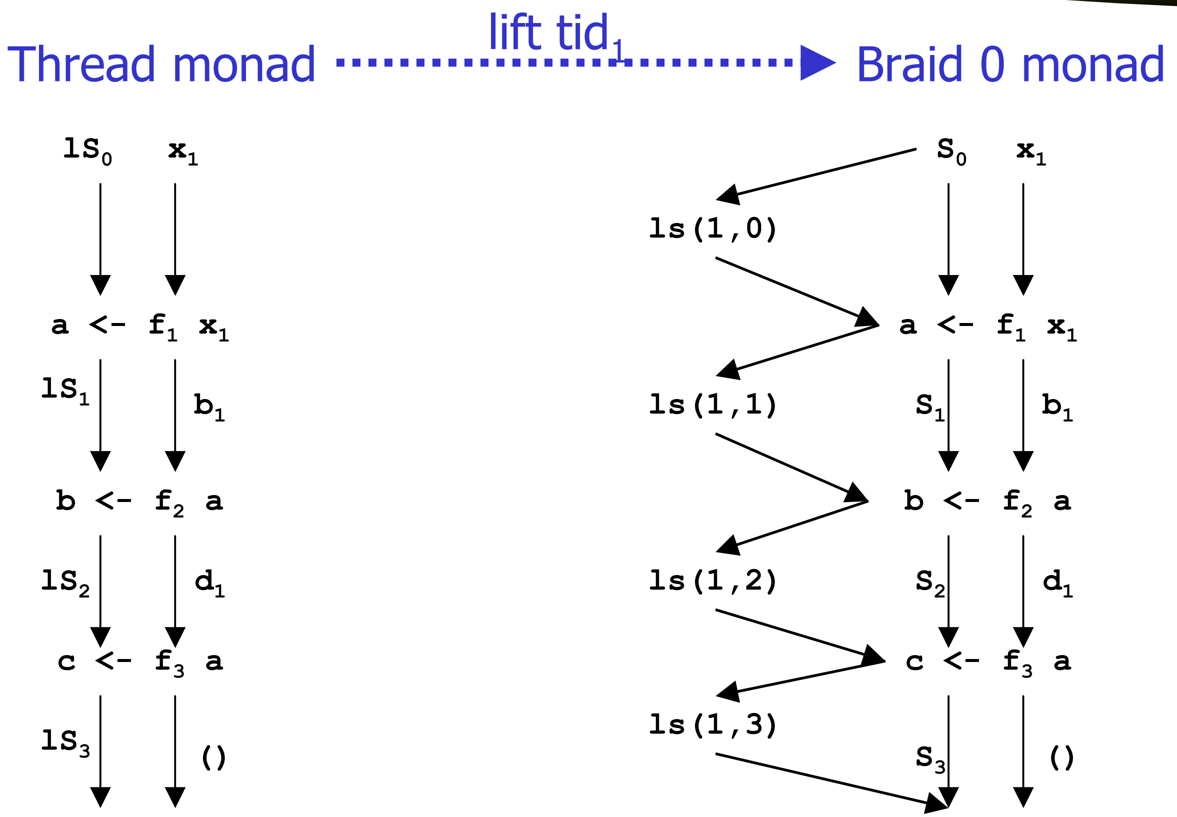
local program	tid_1 prog(1,1)	tid_2 ,1)	tid_3 ,1)
local state	tid_1 ls(1,1)	tid_2 ls(2,1)	tid_3 ls(3,1)
queues	A->B [y]	B->C [w]	
threadstate	tid_1 running	tid_2 ready	tid_3 ready
current	tid_1		

sendQ d_1 y

local program	tid_1 prog(1,1)	tid_2 ,1)	tid_3 ,1)
local state	tid_1 ls(1,1)	tid_2 ls(2,1)	tid_3 ls(3,1)
queues	A->B [y]	B->C [w]	
threadstate	tid_1 running	tid_2 ready	tid_3 ready
current	tid_1		

- Local program has type Braid s ()
  - Braid is a recursive data type

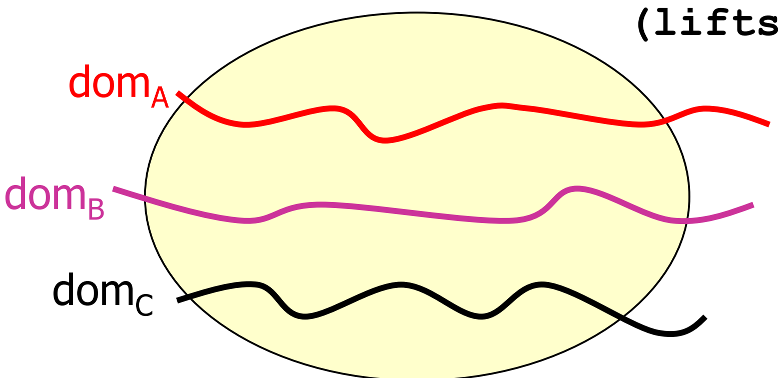
Lifting into the braid 0 monad



Lifting an isolated thread into the braid results in a computation that runs as if it were isolated within the context of the braid.

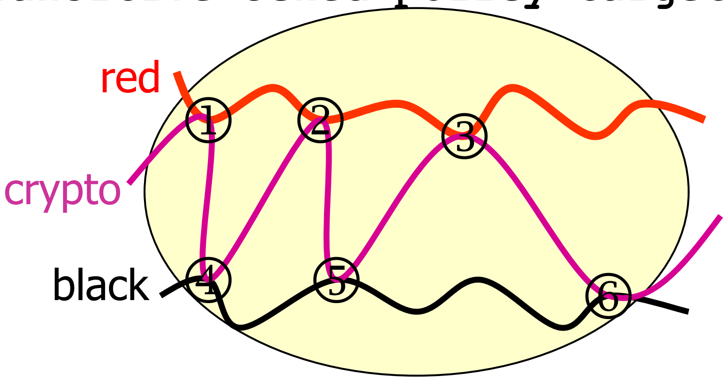
Transitive interference

```
{-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 tinits :: [TH.ThreadInit (Thread ())].
        { filterObservationsTransitive
            sched filt (observations sched (lifts bst tinits))
        } ===
        { observations (filterScheduleTransitive sched filt)
                       (lifts bst tinits) }
```



Intransitive interference

```
{-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 tinits :: [TH.ThreadInit (Thread ())].
        { filterObservationsIntransitive
            policy target (observations sched (lifts bst tinits))
        } ===
        { observations (filterScheduleIntransitive sched policy target)
                       (lifts bst tinits)
        }
```



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 >> getTidState tid2 } ===
          { getTidState 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 our for lifted threads)

local program	tid_1 prog(1,1)	tid_2 ,1)	tid_3 ,1)
local state	tid_1 ls(1,1)	tid_2 ls(2,1)	tid_3 ls(3,1)
queues	A->B [y]	B->C [w]	
threadstate	tid_1 running	tid_2 ready	tid_3 ready
current	tid_1		

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)

Structure	Separation / process	Separation / kernel	Adequacy	Swept Under Rug	Performance	Mutable State	Exceptions	Interleaving	Kernel Calls / Exec
Thread monad	3	1	1	5	5	Y	N	N	N
Thread Braid 0 / unlifted	3	1	2	5	5	Y	N	N	Y
local / lifted	3	3	2	5	5	Y	N	N	Y

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

Structure	Separation / process	Separation / kernel	Adequacy	Swept Under Rug	Performance	Mutable State	Exceptions	Interleaving	Kernel Calls / Exec
Thread monad	3	1	1	5	5	Y	N	N	N
Thread Braid 0 / unlifted	3	1	2	5	5	Y	N	N	Y
local / lifted	3	3	2	5	5	Y	N	N	Y
Braid 1	5	3	3	5	5	Y	Y	N	Y

State evolution in braid 1

- 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

local catcher	tid_1 catch(1,1)	tid_2)	tid_3
local program	tid_1 prog(1,1)	tid_2 ,1)	tid_3 ,1)
local state	tid_1 ls(1,1)	tid_2 ls(2,1)	tid_3 ls(3,1)
queues	A->B [y]	B->C [w]	
threadstate	tid_1 running	tid_2 ready	tid_3 ready
current	tid_1		

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

local catcher	tid$_1$ catch(1,1)	tid$_2$)	tid$_3$
local program	tid$_1$ prog(1,1)	tid$_2$,1)	tid$_3$,1)
local state	tid$_1$ ls(1,1)	tid$_2$ ls(2,1)	tid$_3$ ls(3,1)
queues	A->B [y]	B->C [w]	
threadstate	tid$_1$ running	tid$_2$ ready	tid$_3$ ready
current	tid$_1$		

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

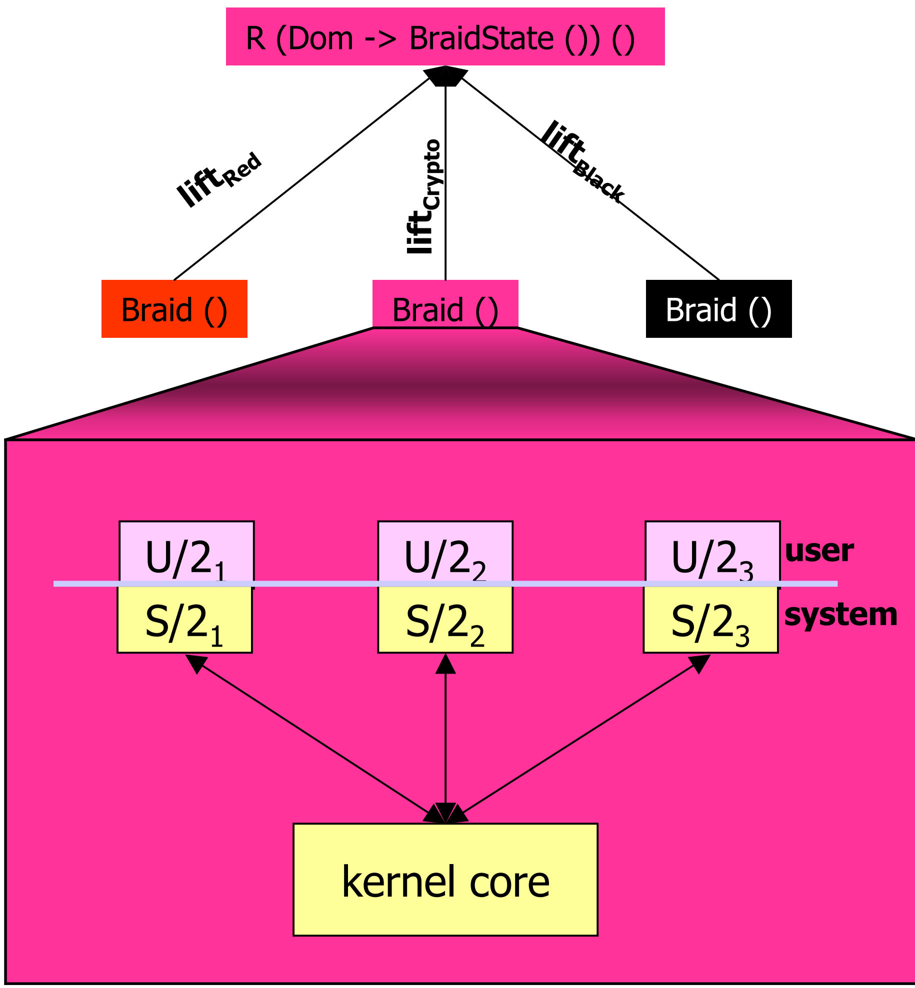
Structure	Separation / process	Separation / kernel	Adequacy	Swept Under Rug	Performance	Mutable State	Exceptions	Interleaving	Kernel Calls / Exec
Thread monad	3	1	1	5	5	Y	N	N	N
Thread Braid 0 / unlifted	3	1	2	5	5	Y	N	N	Y
local / lifted	3	3	2	5	5	Y	N	N	Y
Braid 1	5	3	3	5	5	Y	Y	N	Y
Braid 2	5	3	5	5	5	Y	Y	Y	Y

Separation in braid 2

- The separation properties in braid 2 are identical to those in braid 0

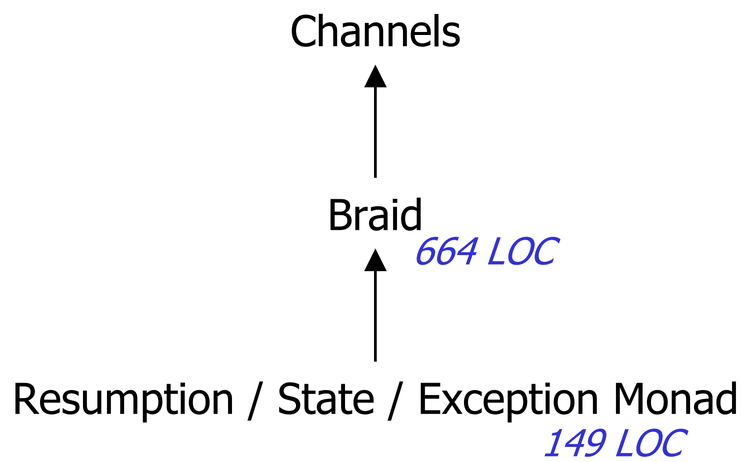
The end game: Process separation

- 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



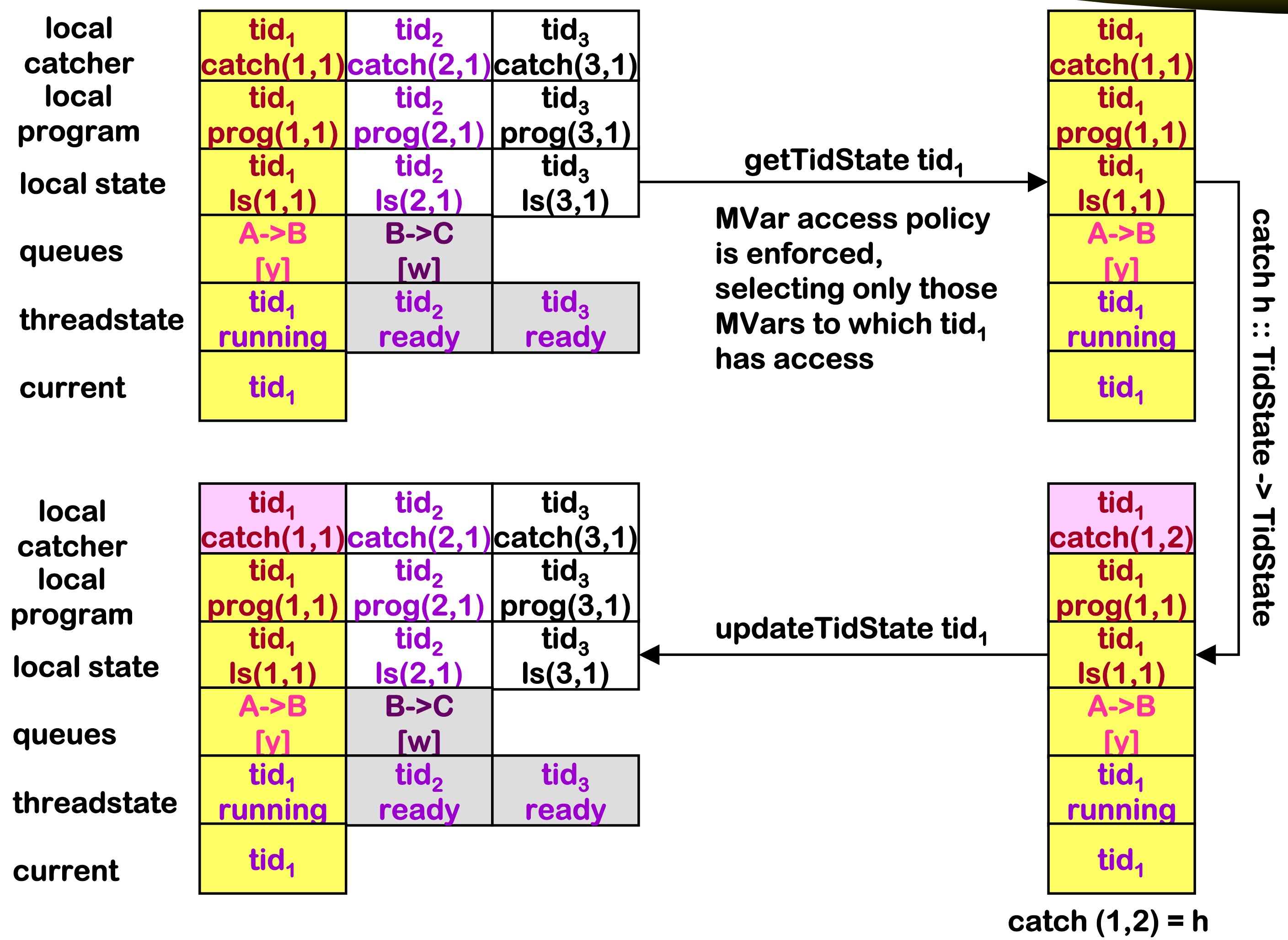
Taming kernel calls

- 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




```
readMVar       newEmptyMVar   MVar
yield          newMVar        Exception
fork           deleteMVar     ThreadId
killThread     takeMVar       Thread
threadDelay    putMVar
myThreadId     modifyMVar
throw          withMVar
throwTo        swapMVar
catch          weave
```

Structured and restricted kernel calls



Structured kernel calls

- Another way to look at it: Refactoring

f tid a b c = f' a b c (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

Structure	Separation / process	Separation / kernel	Adequacy	Swept Under Rug	Performance	Mutable State	Exceptions	Interleaving	Kernel Calls / Exec
Thread monad	3	1	1	5	5	Y	N	N	N
Thread Braid 0 / unlifted	3	1	2	5	5	Y	N	N	Y
local / lifted	3	3	2	5	5	Y	N	N	Y
Braid 1	5	3	3	5	5	Y	Y	N	Y
Braid 2	5	3	5	5	5	Y	Y	Y	Y
Structured Kernel Calls	5	4	5	5	5	Y	Y	Y	Y

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)