Lightweight Concurrency in GHC

KC Sivaramakrishnan
Tim Harris
Simon Marlow
Simon Peyton Jones

GHC: Concurrency and Parallelism

forkIO

MVars

Safe foreign calls

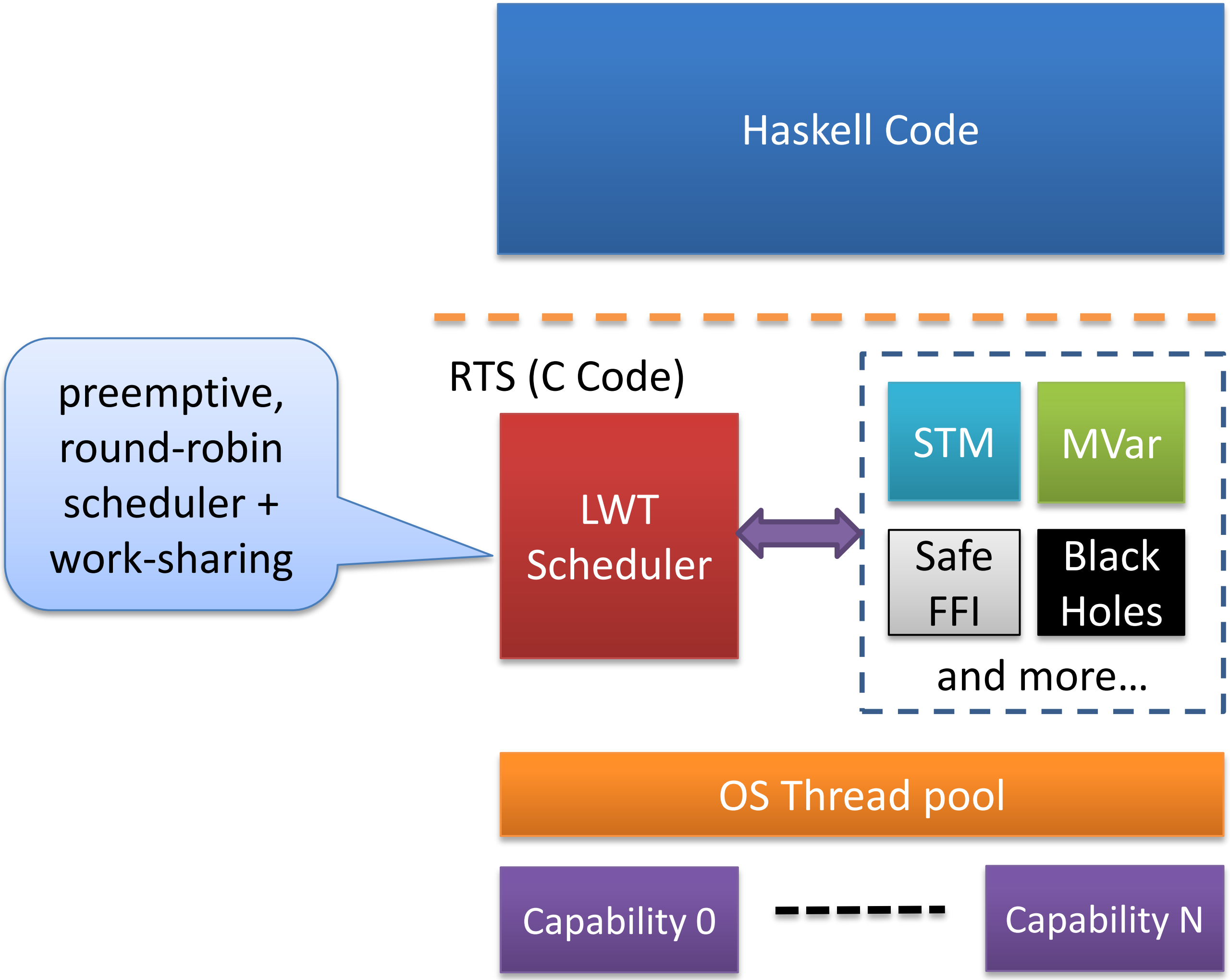
Bound threads

Par Monad

STM

Asynchronous exceptions

Concurrency landscape in GHC



Haskell Code

RTS (C Code)

preemptive, round-robin scheduler + work-sharing

LWT Scheduler

STM

MVar

Safe FFI

Black Holes

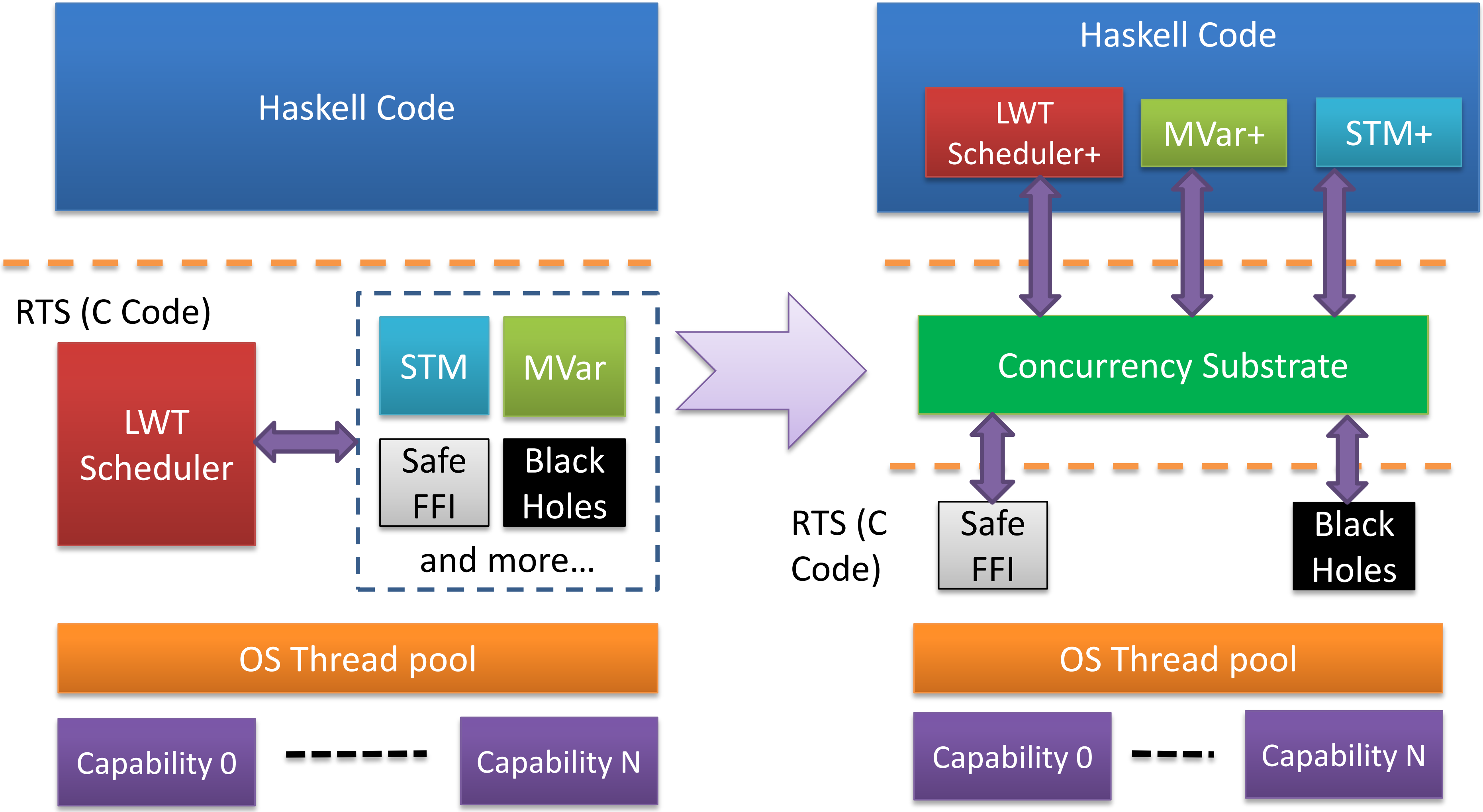
and more…

OS Thread pool

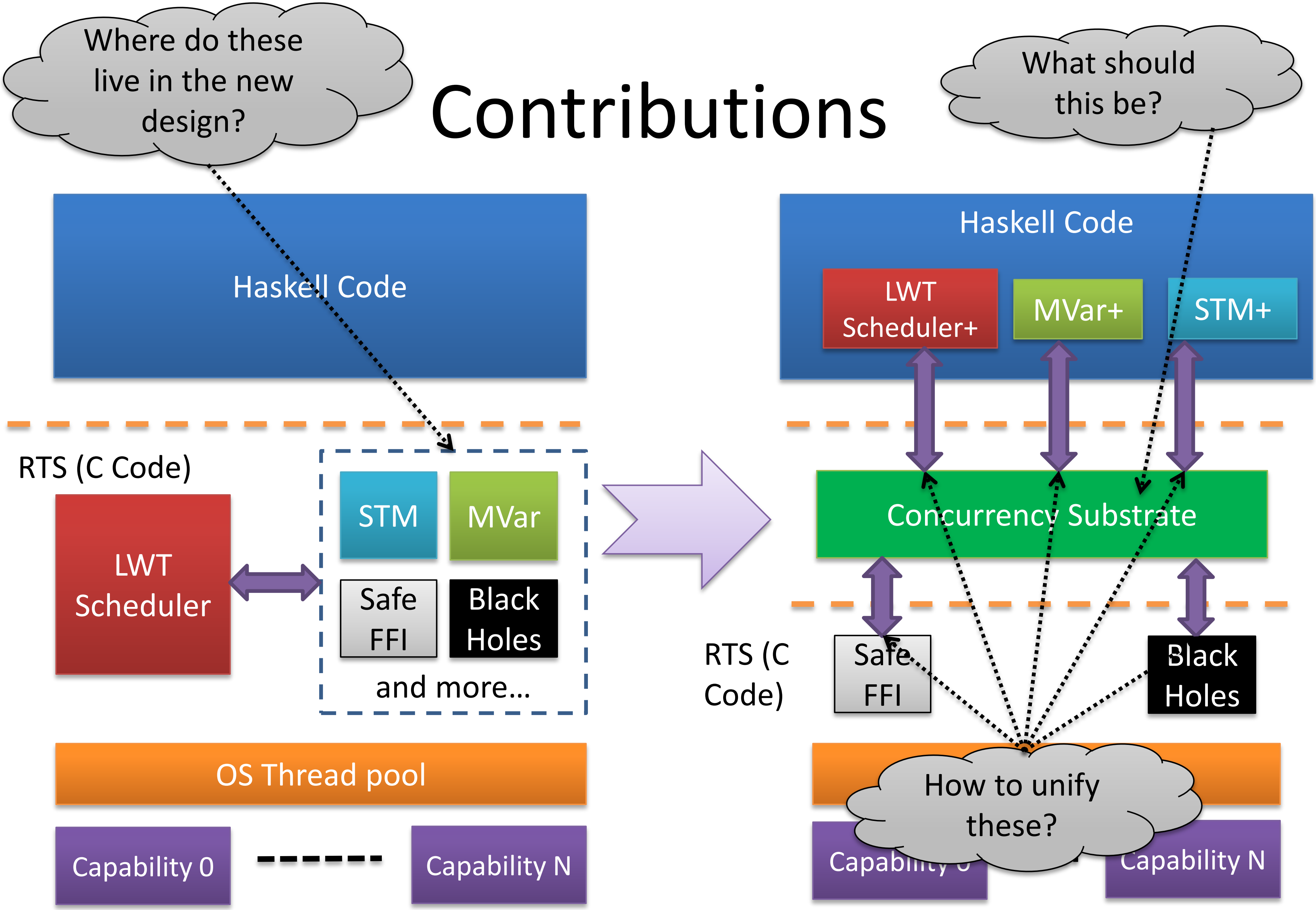
Capability 0

Capability N

Idea


Haskell Code
RTS (C Code)
LWT Scheduler
STM
MVar
Safe FFI
Black Holes
and more…
OS Thread pool
Capability 0
Capability N
Haskell Code
LWT Scheduler+
MVar+
STM+
Concurrency Substrate
RTS (C Code)
Safe FFI
Black Holes
OS Thread pool
Capability 0
Capability N

Contributions



Where do these live in the new design?

What should this be?

Haskell Code

RTS (C Code)

LWT Scheduler

STM

MVar

Safe FFI

Black Holes

and more…

OS Thread pool

Capability 0

Capability N

Haskell Code

LWT Scheduler+

MVar+

STM+

Concurrency Substrate

RTS (C Code)

Safe FFI

Black Holes

How to unify these?

Capability 0

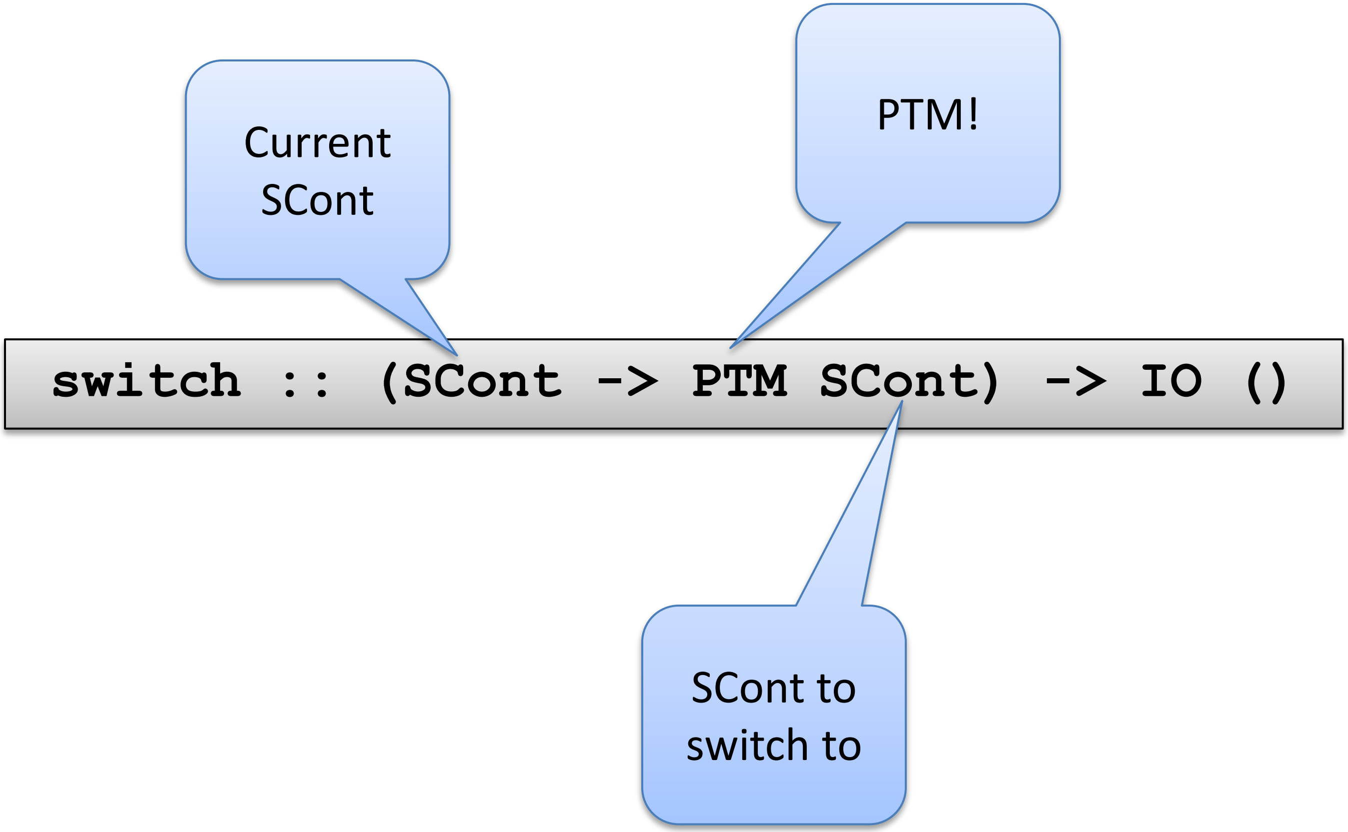
Capability N

Concurrency Substrate

- One-shot continuations (SCont) and primitive transactional memory (PTM)
- PTM is a bare-bones TM
  - Better composability than CAS

```
---------------- PTM ----------------

data PTM a
data PVar a
instance Monad PTM

atomically :: PTM a -> IO a
newPVar    :: a -> PTM (PVar a)
readPVar   :: PVar a -> PTM a
writePVar  :: PVar a -> a -> PTM ()

---------------- SCont --------------

data SCont -- Stack Continuations
newSCont :: IO () -> IO SCont
switch   :: (SCont -> PTM SCont) -> IO ()
getCurrentSCont :: PTM SCont
switchTo :: SCont -> PTM ()
```

Switch



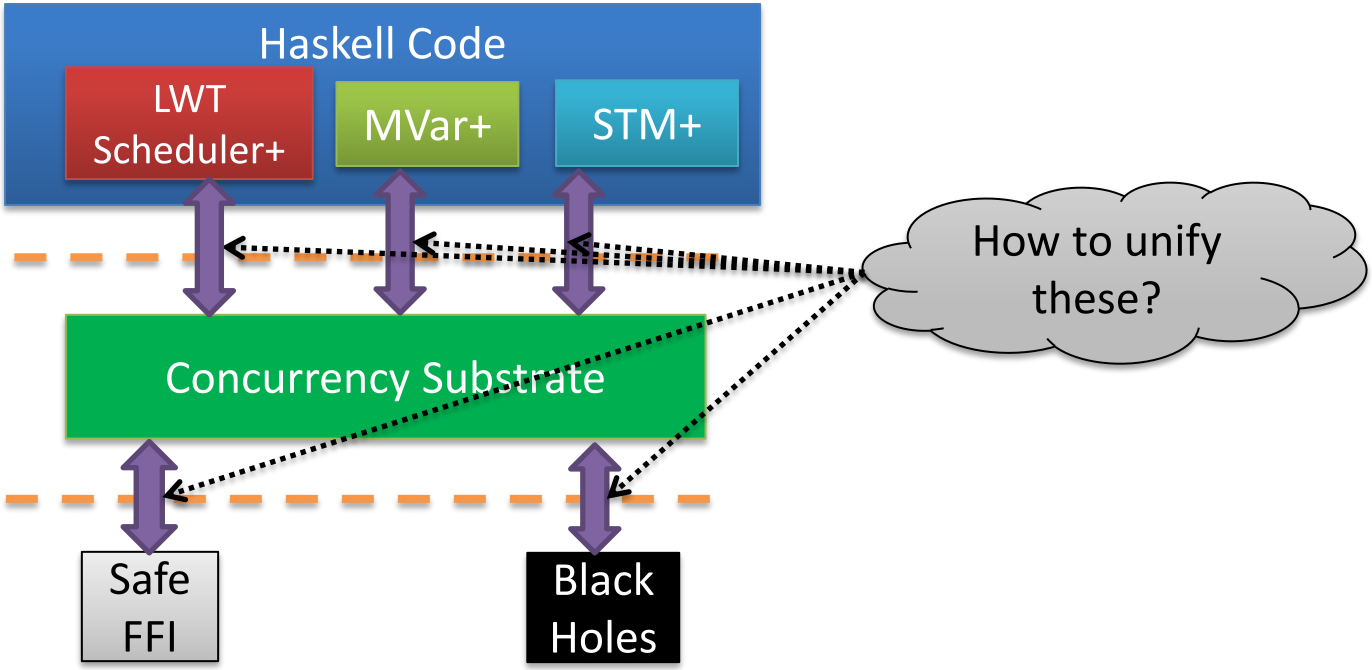
Current SCont

PTM!

```
switch :: (SCont -> PTM SCont) -> IO ()
```

SCont to switch to

Abstract Scheduler Interface



- Primitive scheduler actions
  - SCont {**scheduleSContAction** :: SCont -> PTM (),
    yieldControlAction :: PTM ()}
  - Expected from every user-level thread

Primitive Scheduler Actions (1)

```
scheduleSContAction :: SCont -> PTM ()
scheduleSContAction sc = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  writePVar $ contents ++ [sc]
```

```
yieldControlAction :: PTM ()
yieldControlAction = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  case contents of
    x:tail -> do {
      writePVar $ contents tail;
      switchTo x -- DOES NOT RETURN
    }
    otherwise -> …
```

Primitive Scheduler Actions (2)

```
scheduleSContAction :: SCont -> PTM ()
scheduleSContAction sc = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  writePVar $ contents ++ [sc]
```

```
yieldControlAction :: PTM ()
yieldControlAction = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  case contents of
    x:tail -> do {
      writePVar $ contents tail;
      switchTo x -- DOES NOT RETURN
    }
    otherwise -> …
```

Substrate Primitives

```
getScheduleSContAction
  :: SCont -> PTM (SCont -> PTM())
setScheduleSContAction
  :: SCont -> (SCont -> PTM()) -> PTM()
```

```
getYieldControlAction
  :: SCont -> PTM (PTM ())
setScheduleSContAction
  :: SCont -> PTM () -> PTM ()
```

Primitive Scheduler Actions (3)

```
scheduleSContAction :: SCont -> PTM ()
scheduleSContAction sc = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  writePVar $ contents ++ [sc]
```

Substrate Primitives

```
getScheduleSContAction
  :: SCont -> PTM (SCont -> PTM())
setScheduleSContAction
  :: SCont -> (SCont -> PTM()) -> PTM()

getSSA = getScheduleSContAction
setSSA = setScheduleScontAction
```

Helper functions

```
yieldControlAction :: PTM ()
yieldControlAction = do
  sched :: PVar [SCont] <- -- get sched
  contents :: [SCont] <- readPVar sched
  case contents of
    x:tail -> do {
      writePVar $ contents tail;
      switchTo x -- DOES NOT RETURN
    }
    otherwise -> …
```

```
getYieldControlAction
  :: SCont -> PTM (PTM ())
setScheduleSContAction
  :: SCont -> PTM () -> PTM ()

getYCA = getYieldControlAction
setYCA = setYieldControlAction
```

Building Concurrency Primitives (1)

```
yield :: IO ()
yield = atomically $ do
  s :: SCont <- getCurrentSCont
  -- Add current SCont to scheduler
  ssa :: (SCont -> PTM ()) <- getSSA s
  enque :: PTM () <- ssa s
  enque
  -- Switch to next scont from scheduler
  switchToNext :: PTM () <- getYCA s
  switchToNext
```

Building Concurrency Primitives (2)

```
forkIO :: IO () -> IO SCont
forkIO f = do
  ns <- newSCont f
  atomically $ do {
    s :: SCont <- getCurrentSCont;
    -- Initialize new sconts scheduler actions
    ssa :: (SCont -> PTM ()) <- getSSA s;
    setSSA ns ssa;
    yca :: PTM () <- getYCA s;
    setYCA ns yca;
    -- Add to new scont current scheduler
    enqueAct :: PTM () <- ssa ns;
    enqueAct
  }
  return ns
```

Building MVars

```
newtype MVar a = MVar (PVar (ST a))
data ST a = Full a [(a, PTM())]
          | Empty [(PVar a, PTM())]

takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  hole <- atomically $ newPVar undefined
  atomically $ do
    st <- readPVar ref
    case st of
      Empty ts -> do
        s <- getCurrentSCont
        ssa :: (SCont -> PTM ()) <- getSSA s
        wakeup :: PTM () <- ssa s
        writePVar ref $ v
          where v = Empty $ ts++[(hole, wakeup)]
        switchToNext <- getYCA s
        switchToNext
      Full x ((x', wakeup :: PTM ()):ts) -> do
        writePVar hole x
        writePVar ref $ Full x' ts
        wakeup
      otherwise -> …
  atomically $ readPVar hole
```

An MVar is either empty or full and has a single hole

Building MVars

```
newtype MVar a = MVar (PVar (ST a))
data ST a = Full a [(a, PTM())]
          | Empty [(PVar a, PTM())]
```

An MVar is either empty or full and has a single hole

```
takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  hole <- atomically $ newPVar undefined
  atomically $ do
    st <- readPVar ref
    case st of
      Empty ts -> do
        s <- getCurrentSCont
        ssa :: (SCont -> PTM ()) <- getSSA s
        wakeup :: PTM () <- ssa s
        writePVar ref $ v
          where v = Empty $ ts++[(hole, wakeup)]
        switchToNext <- getYCA s
        switchToNext
      Full x ((x', wakeup :: PTM ()):ts) -> do
        writePVar hole x
        writePVar ref $ Full x' ts
        wakeup
      otherwise -> …
  atomically $ readPVar hole
```

Result will be here

Building MVars

```
newtype MVar a = MVar (PVar (ST a))
data ST a = Full a [(a, PTM())]
          | Empty [(PVar a, PTM())]

takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  hole <- atomically $ newPVar undefined
  atomically $ do
    st <- readPVar ref
    case st of
      Empty ts -> do
        s <- getCurrentSCont
        ssa :: (SCont -> PTM ()) <- getSSA s
        wakeup :: PTM () <- ssa s
        writePVar ref $ v
          where v = Empty $ ts++[(hole, wakeup)]
        switchToNext <- getYCA s
        switchToNext
      Full x ((x', wakeup :: PTM ()):ts) -> do
        writePVar hole x
        writePVar ref $ Full x' ts
        wakeup
      otherwise -> …
  atomically $ readPVar hole
```

An MVar is either empty or full and has a single hole

Result will be here

If the mvar is empty
1. Append hole & wakeup info to mvar list (*getSSA!*)
2. Yield control to scheduler (*getYCA!*)

Building MVars

```
newtype MVar a = MVar (PVar (ST a))
data ST a = Full a [(a, PTM())]
          | Empty [(PVar a, PTM())]

takeMVar :: MVar a -> IO a
takeMVar (MVar ref) = do
  hole <- atomically $ newPVar undefined
  atomically $ do
    st <- readPVar ref
    case st of
      Empty ts -> do
        s <- getCurrentSCont
        ssa :: (SCont -> PTM ()) <- getSSA s
        wakeup :: PTM () <- ssa s
        writePVar ref $ v
          where v = Empty $ ts++[(hole, wakeup)]
        switchToNext <- getYCA s
        switchToNext
      Full x ((x', wakeup :: PTM ()):ts) -> do
        writePVar hole x
        writePVar ref $ Full x' ts
        wakeup
      otherwise -> …
  atomically $ readPVar hole
```

An MVar is either empty or full and has a single hole

Result will be here

If the mvar is empty

(1) Append hole & wakeup info to mvar list (*getSSA!*)

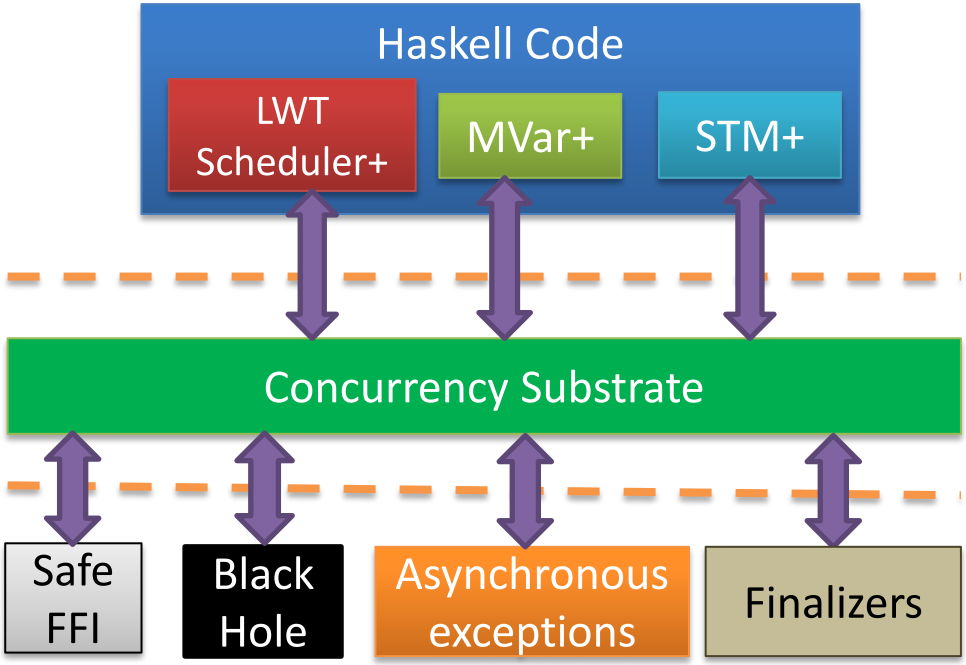
(2) Yield control to scheduler (*getYCA!*)

Wake up a pending writer, if any. wakeup is a PTM ()!

MVar is *scheduler agnostic!*

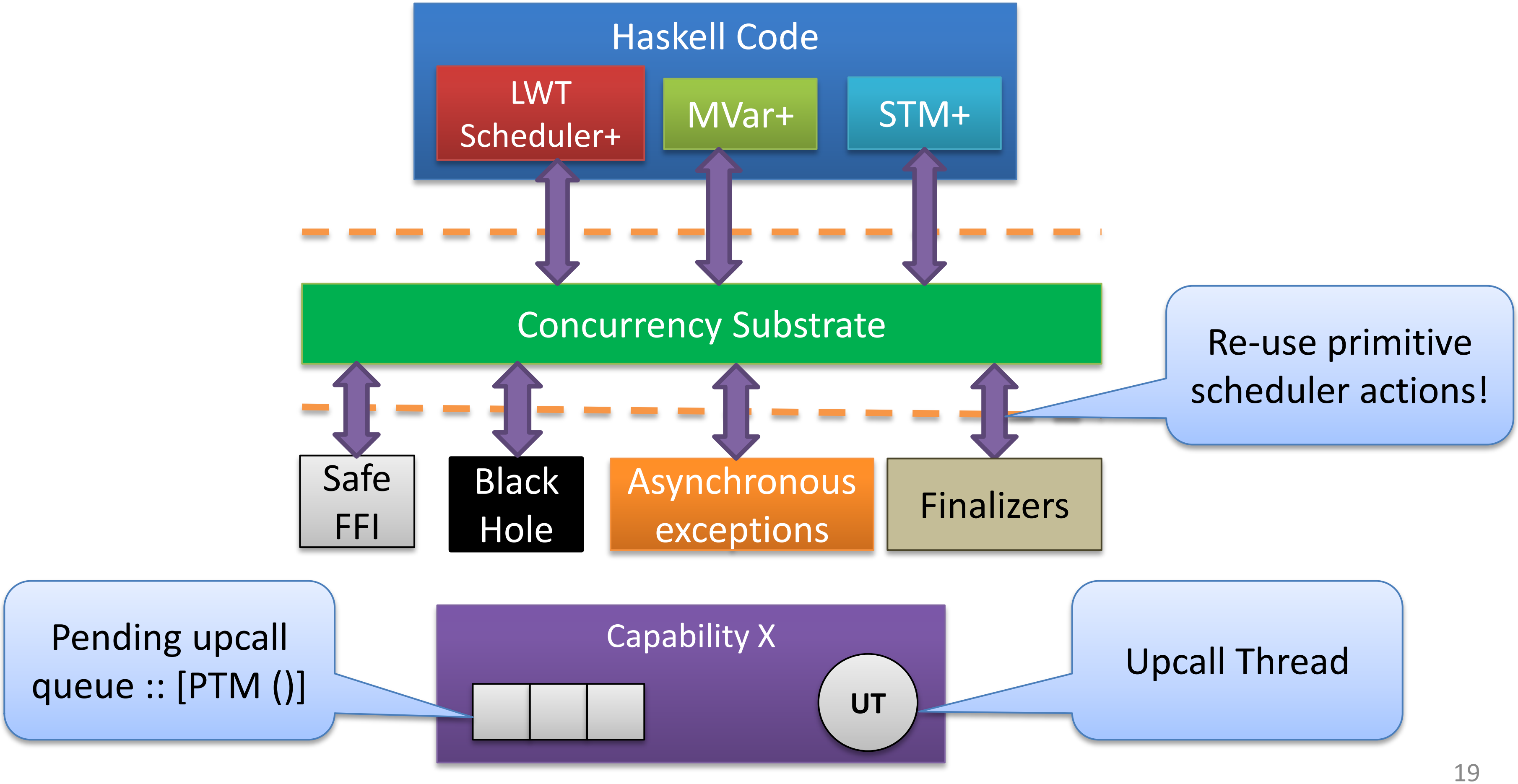
Interaction of C RTS and User-level scheduler

- Many “Events” that necessitate actions on the scheduler become apparent only in the C part of the RTS



Interaction of C RTS and User-level scheduler

- Many “Events” that necessitate actions on the scheduler become apparent only in the C part of the RTS

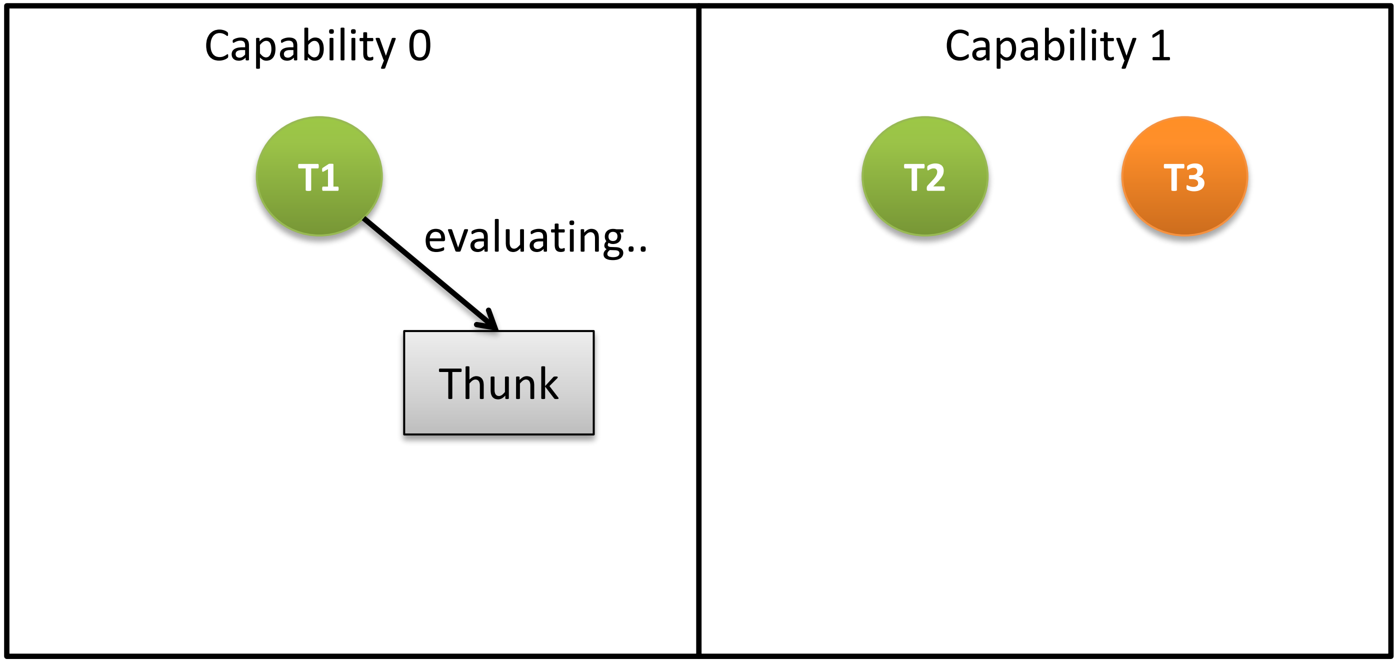


Re-use primitive scheduler actions!

Pending upcall queue :: [PTM ()]

Upcall Thread

Blackholes

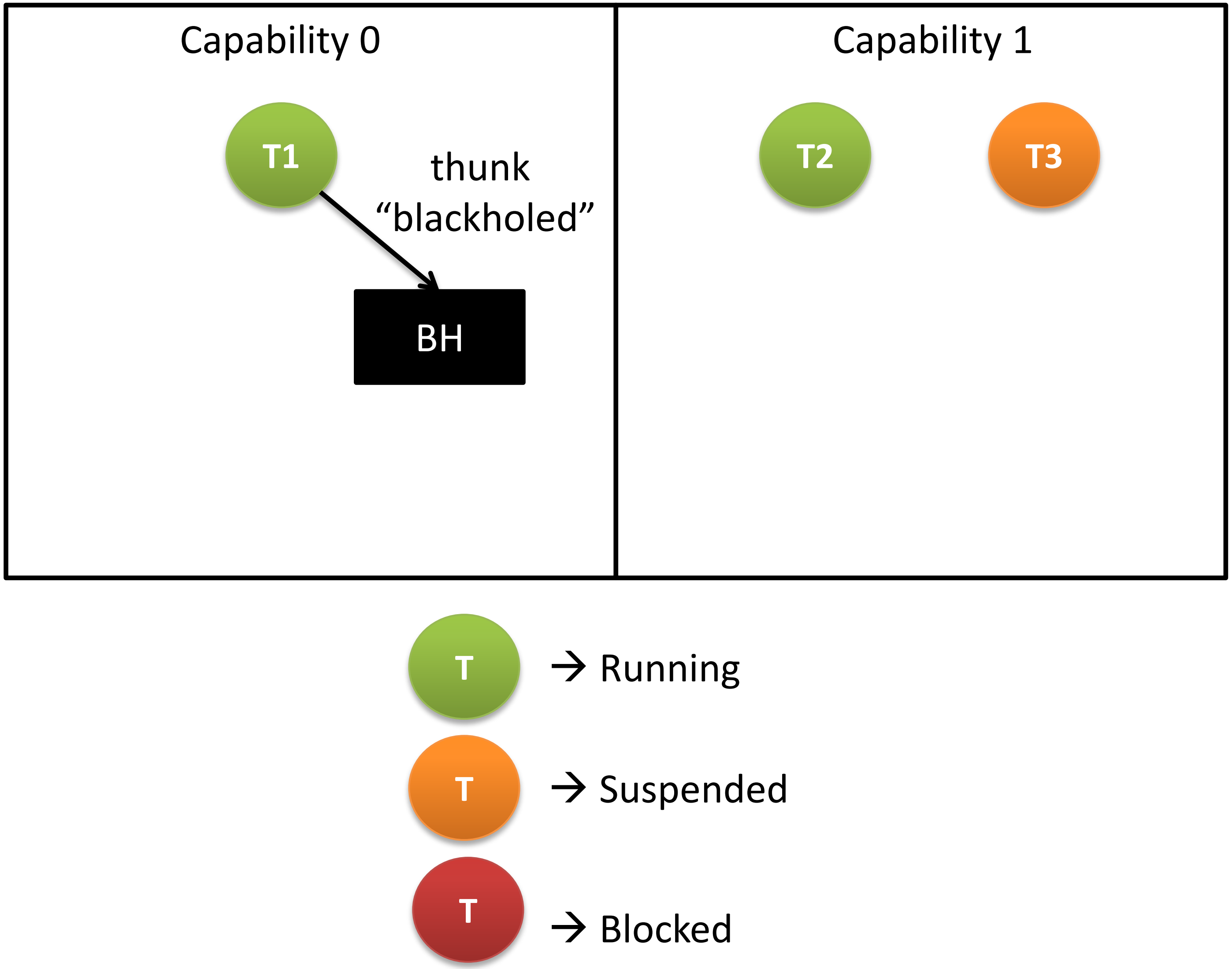


T → Running

T → Suspended

T → Blocked

Blackholes

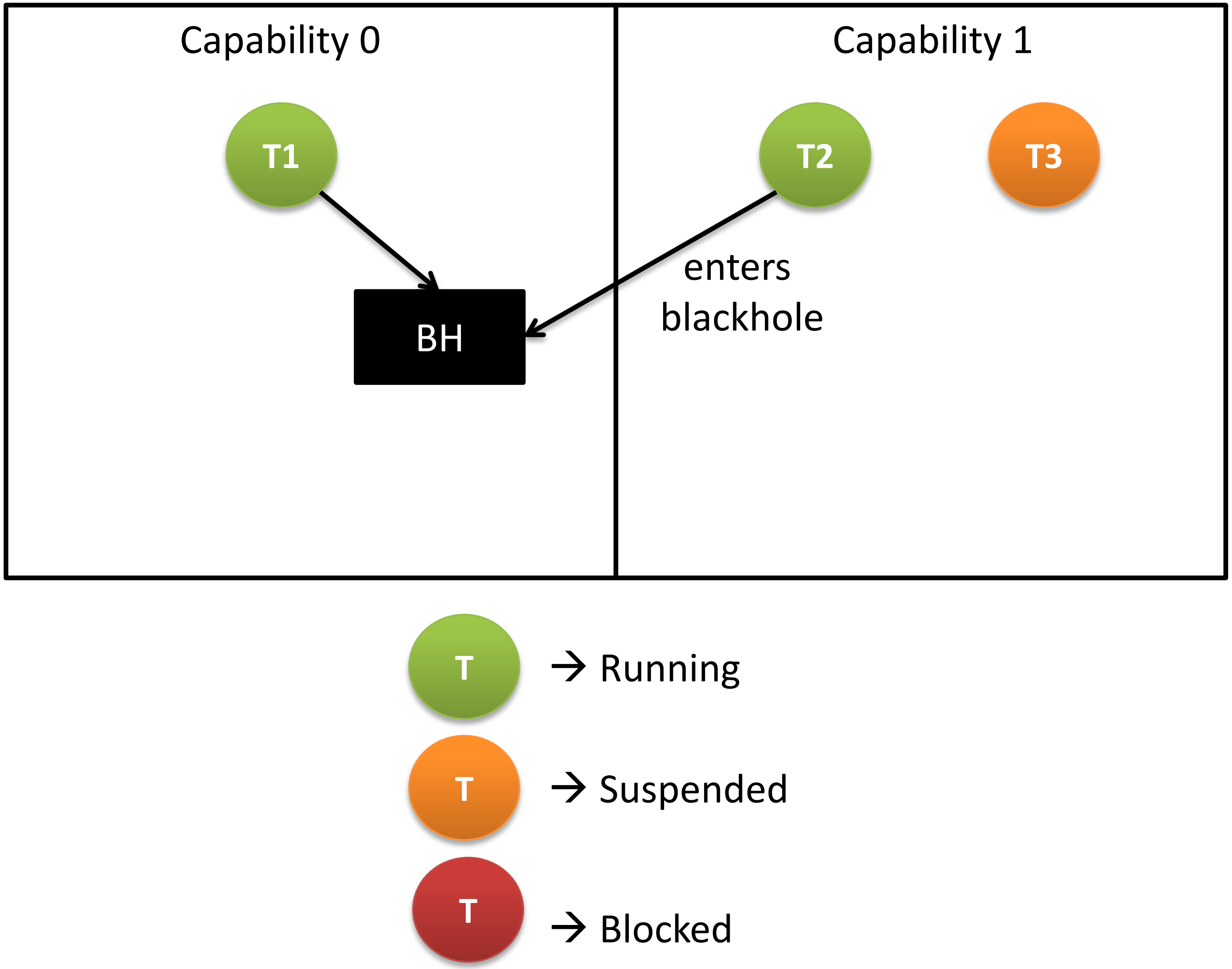


→ Running

→ Suspended

→ Blocked

Blackholes



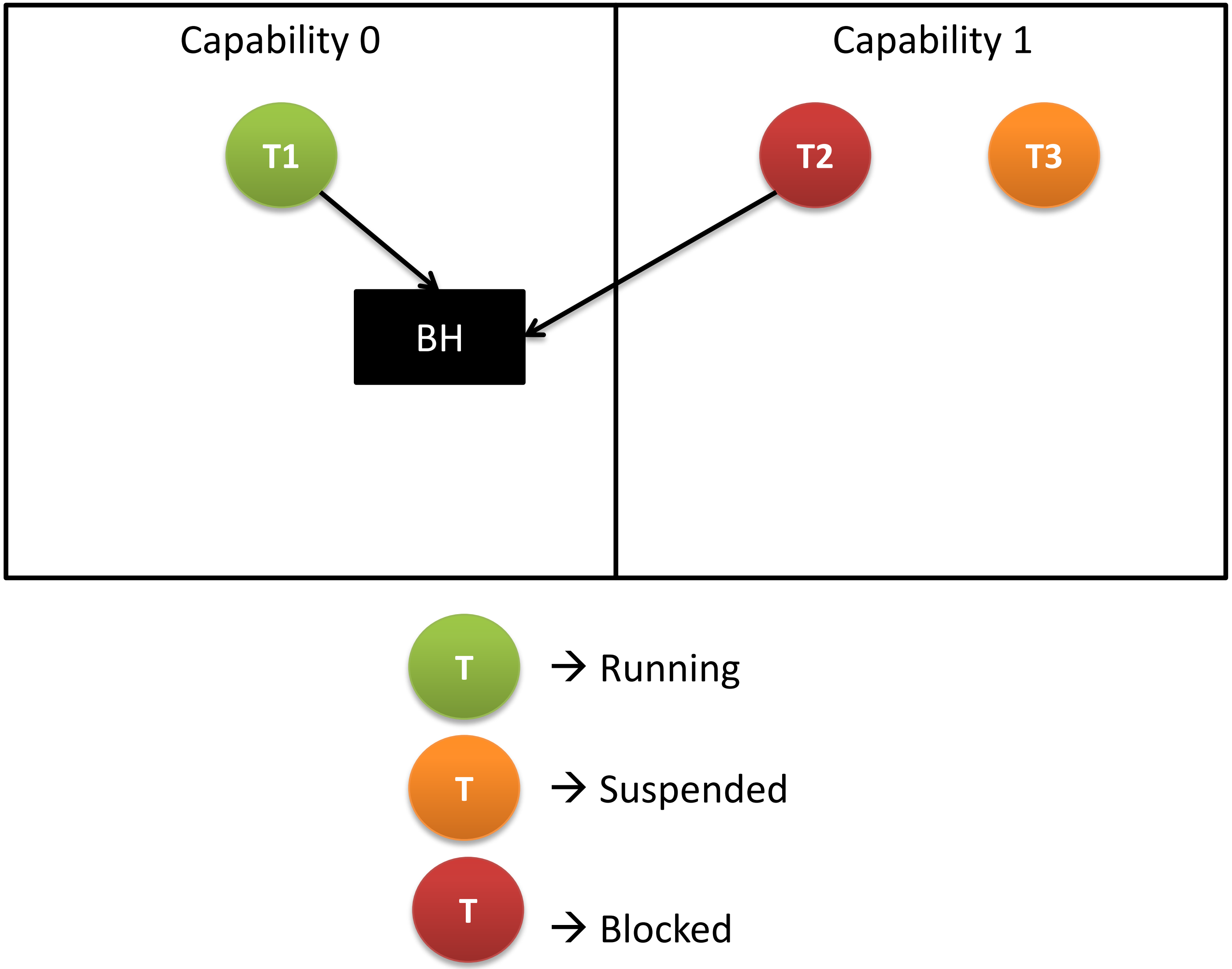
enters blackhole

→ Running

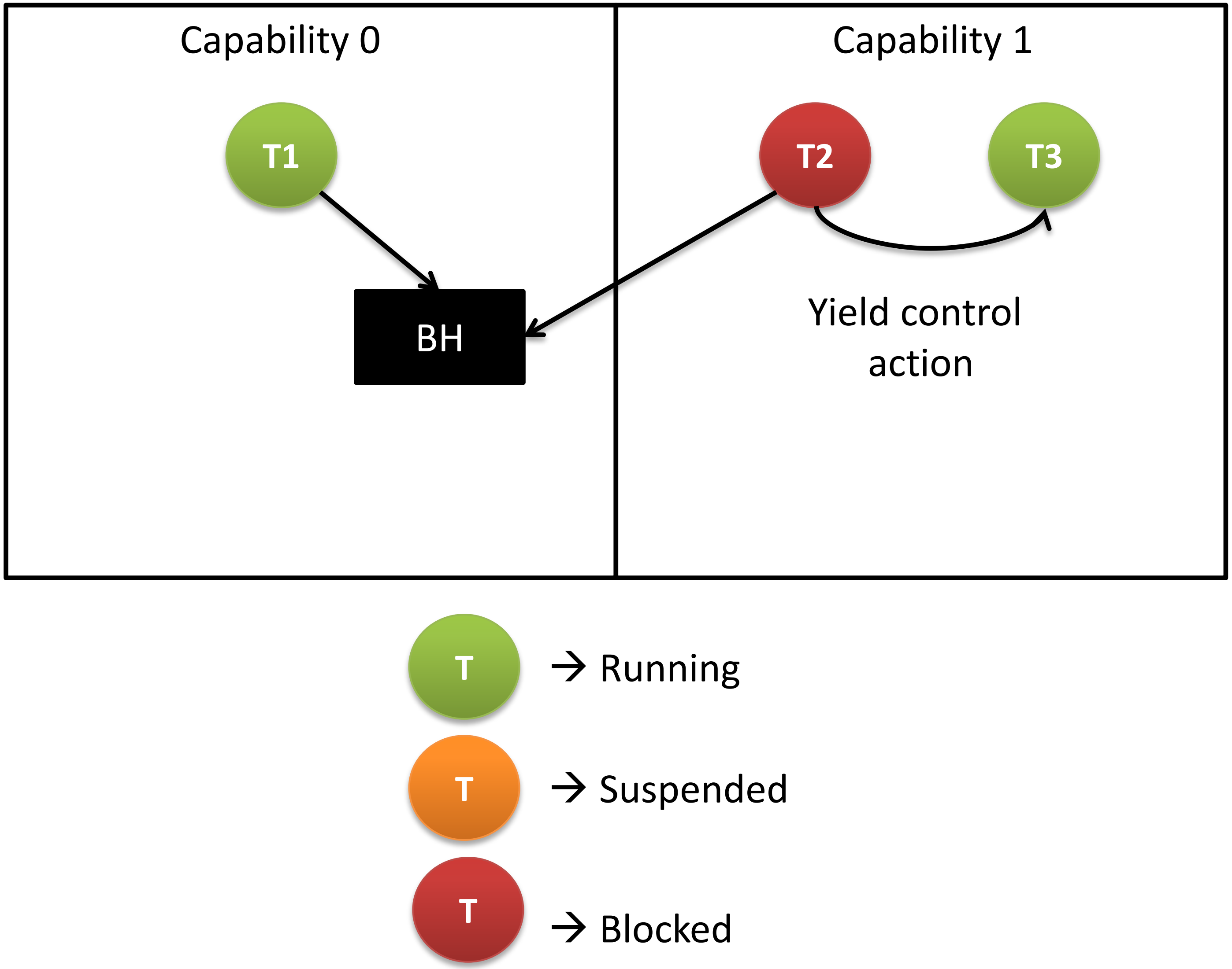
→ Suspended

→ Blocked

Blackholes


Capability 0
Capability 1
T1
T2
T3
BH
T → Running
T → Suspended
T → Blocked

Blackholes



Capability 0

Capability 1

T1

BH

T2

T3

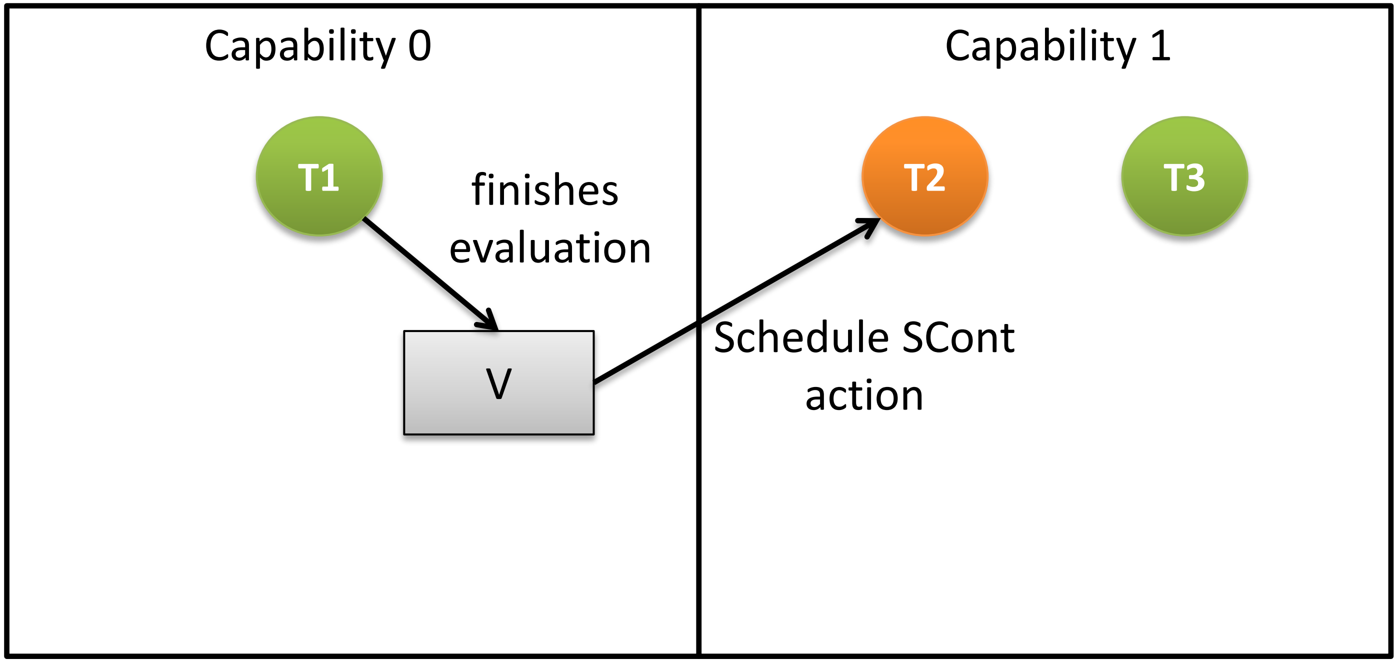
Yield control action

T → Running

T → Suspended

T → Blocked

Blackholes



Capability 0

Capability 1

T1

finishes evaluation

V

T2

T3

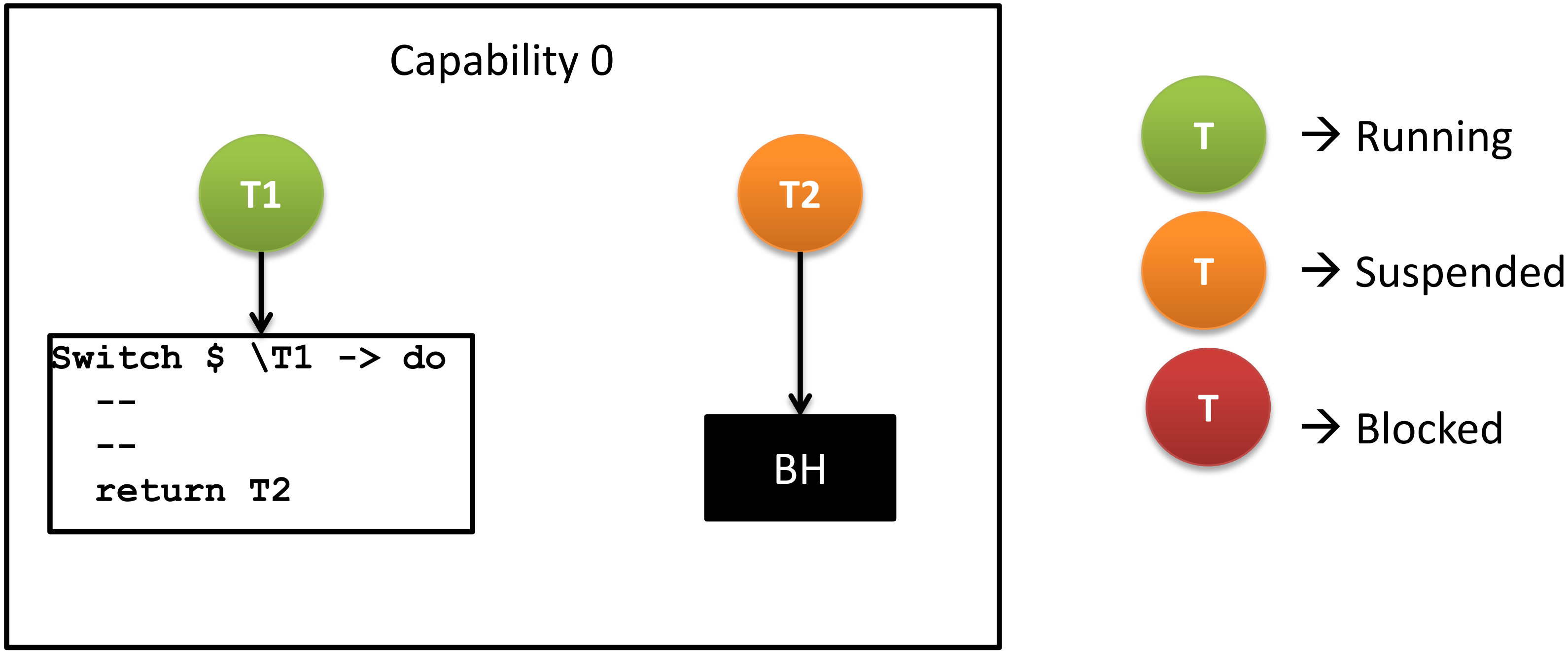
Schedule SCont action

T → Running

T → Suspended

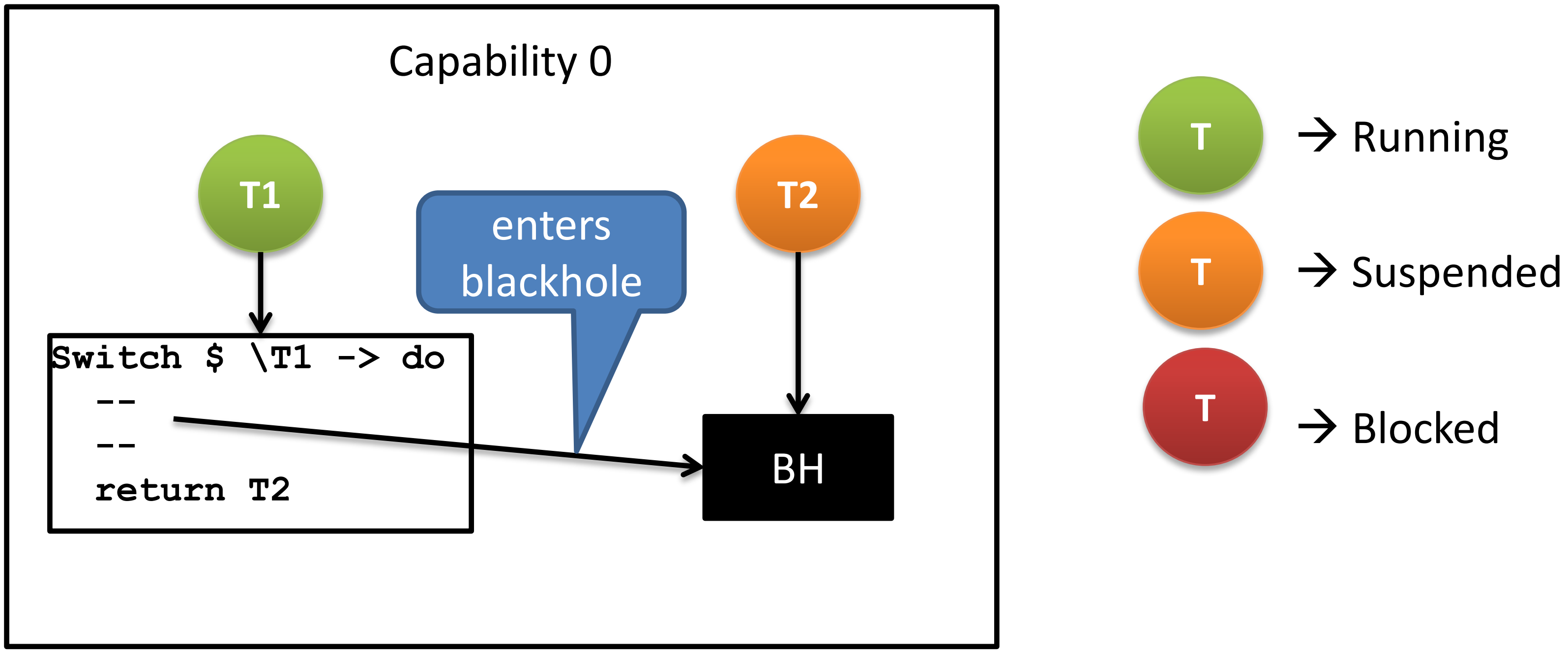
T → Blocked

Blackholes : The Problem



```
Switch $ \T1 -> do
  --
  --
  return T2
```

Blackholes : The Problem



```
Switch $ \T1 -> do
  --
  --
  return T2
```

- In order to make progress, we need to resume to T2
- But, in order to resume to T2, we need to resume T2 (Deadlocked!)
  - Can be resolved through runtime system tricks (Work in Progress!)

Conclusions

- Status
  - Mostly implemented (SConts, PTM, Simple schedulers, MVars, Safe FFI, bound threads, asynchronous exceptions, finalizers, etc.)
  - 2X to 3X slower on micro benchmarks (programs only doing synchronization work)
- To-do
  - Re-implement Control.Concurrent with LWC
  - Formal operational semantics
  - Building real-world programs
- Open questions
  - Hierarchical schedulers, Thread priority, load balancing, Fairness, etc.
  - STM on top of PTM
  - PTM on top of SpecTM
  - Integration with par/seq, evaluation strategies, etc.
  - and more…