GHC(STG,Cmm,asm) illustrated

for hardware persons

exploring some mental models and implementations

Takenobu T.

Rev. 0.01.1
WIP

"Any sufficiently advanced technology is indistinguishable from magic."

Arthur C. Clarke

NOTE

- This is not an official document by the ghc development team.
- Please don't forget “semantics”. It's very important.
- This is written for ghc 7.8 (and ghc 7.10).

Contents

- Executable binary
- Compile steps
- Runtime System
- Development languages

- Machine layer/models
- STG-machine
- Heap objects in STG-machine
- STG-machine evaluation
- Pointer tagging
- Thunk and update
- Allocate and free heap objects
- STG - C land interface

- Thread
- Thread context switch
- Creating main and sub threads
- Thread migration
- Heap and Threads
- Threads and GC
- Bound thread

- Spark

- Mvar
- Software transactional memory

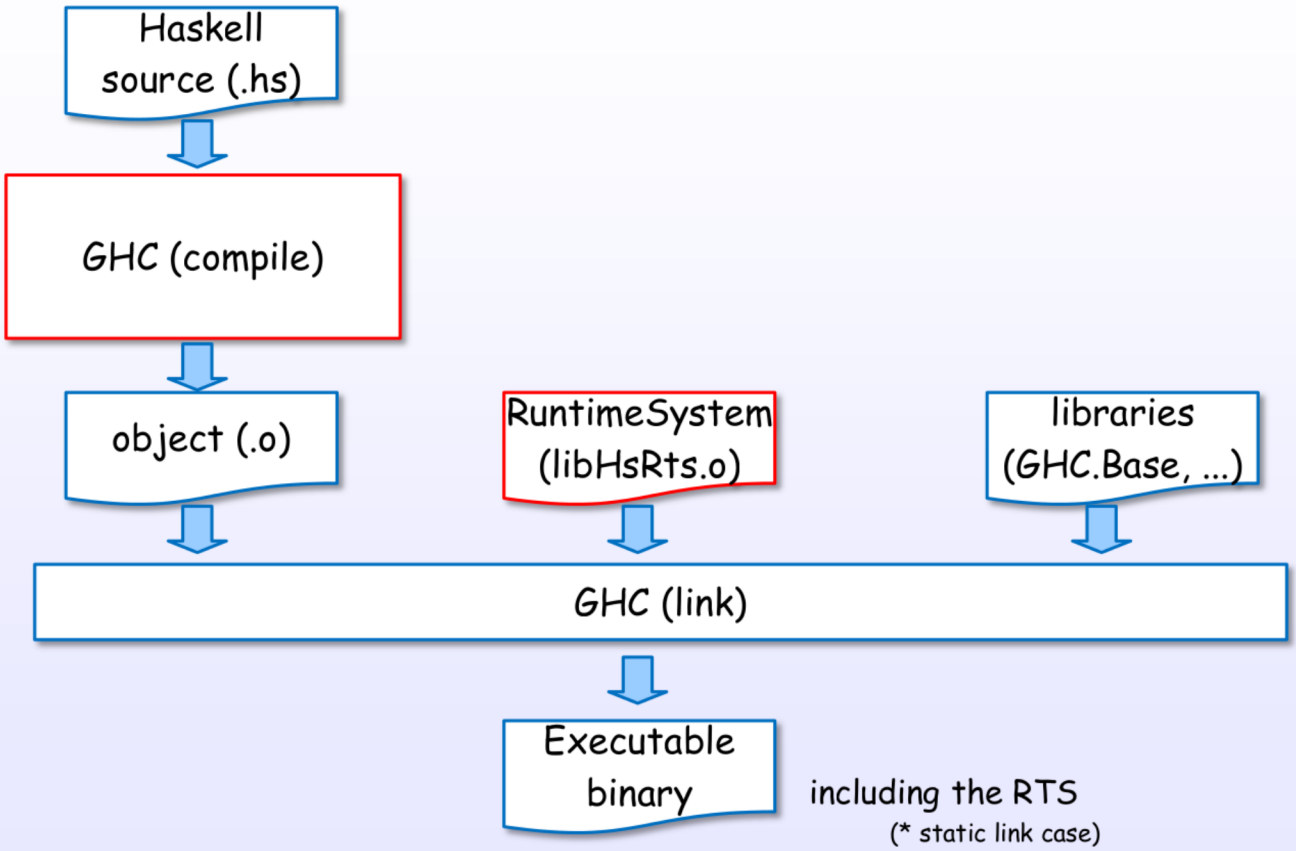
- FFI
- IO and FFI
- IO manager

- Bootstrap

- References

Executable binary

The GHC = Compiler + Runtime System (RTS)

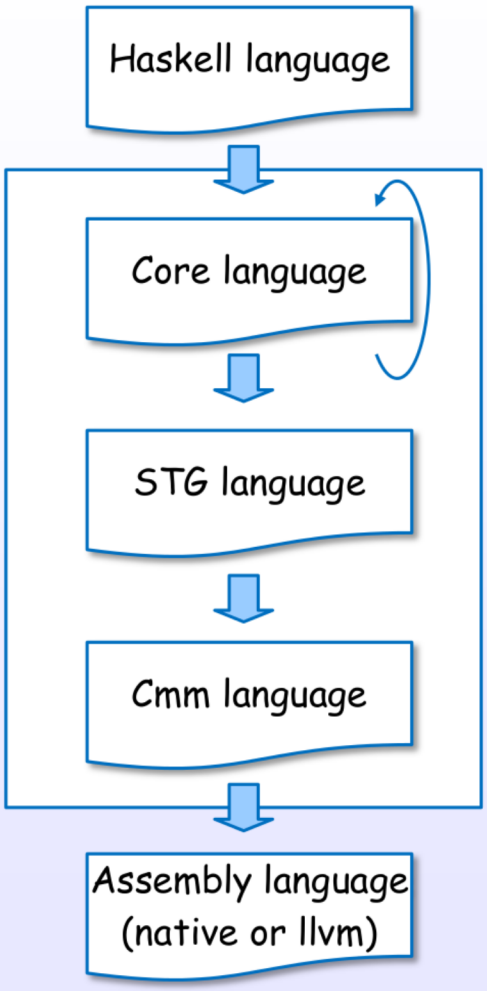


References : [1], [C1], [C3], [C10], [C19], [S7]

Compile steps

GHC transitions between five representations

GHC compile steps



each intermediate code can be dumped by :

% ghc -ddump-parsed
% ghc -ddump-rn

% ghc -ddump-ds
% ghc -ddump-simpl
% ghc -ddump-prep

% ghc -ddump-stg

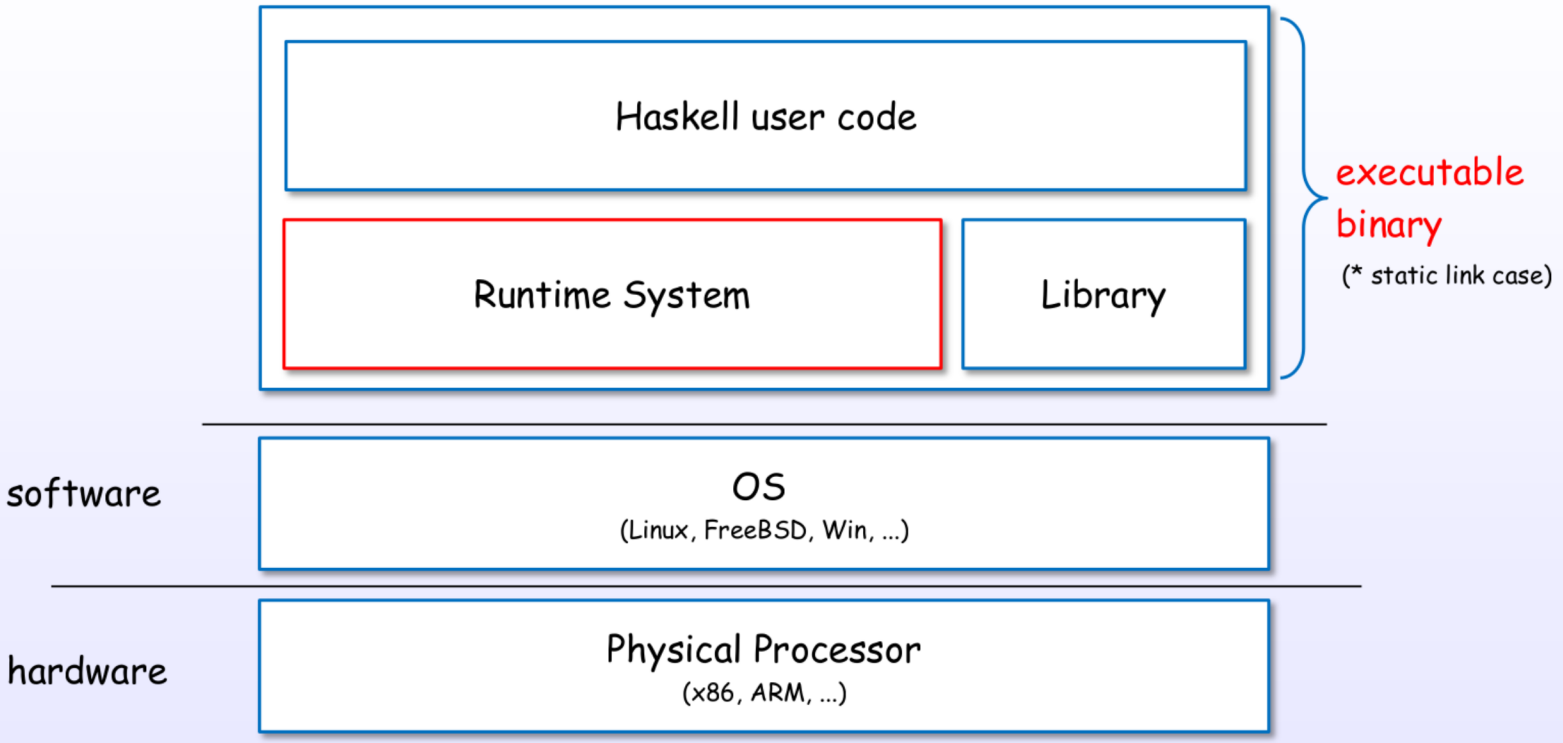
% ghc -ddump-cmm
% ghc -ddump-opt-cmm

% ghc -ddump-llvm
% ghc -ddump-asm

References : [1], [C3], [C4], [9], [C5], [C6], [C7], [C8[], [S7], [S8]

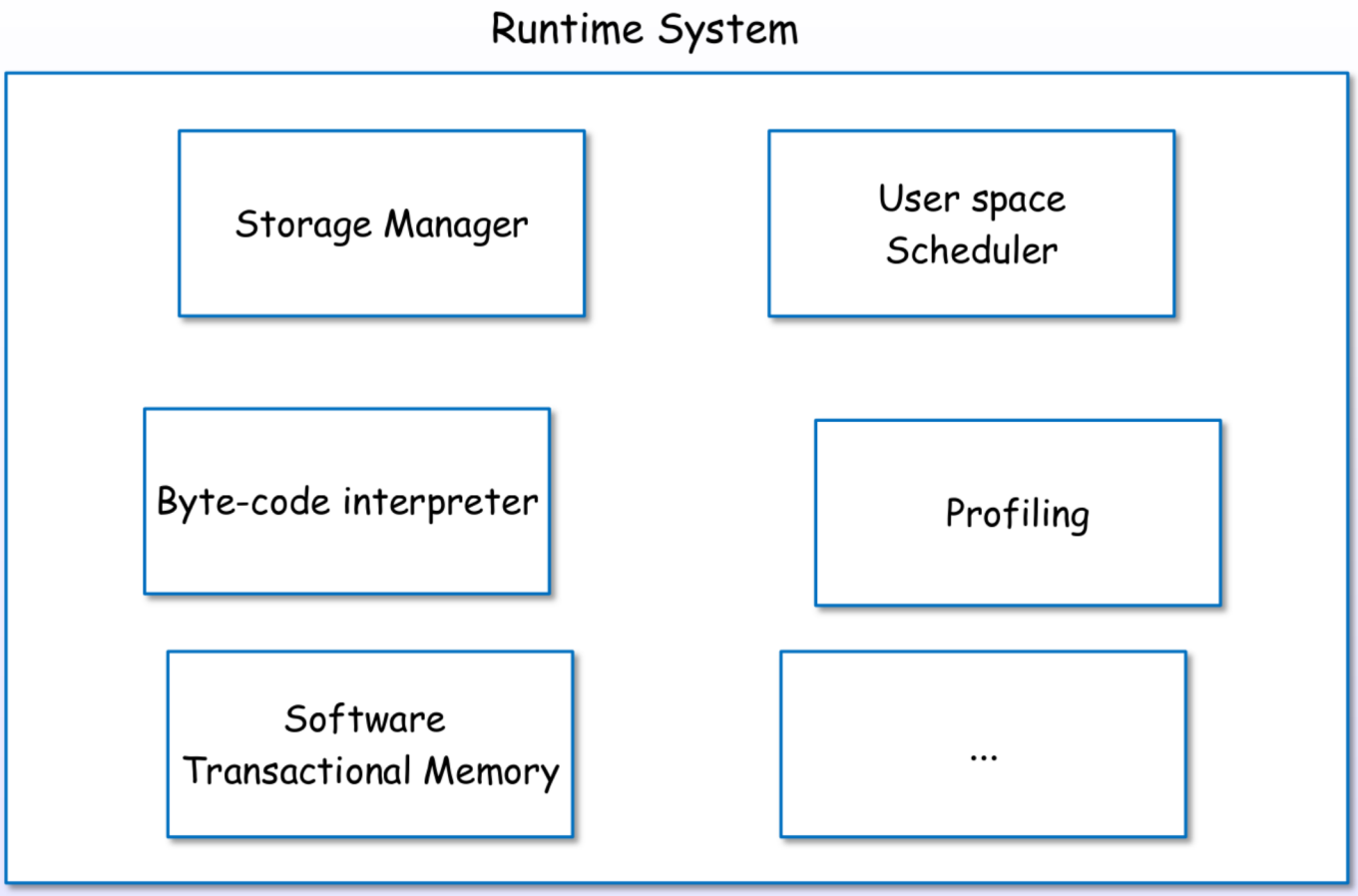
Runtime System

Generated binary includes the RTS



References : [C10], [9]

Runtime System includes ...



References : [C10], [8], [9], [5], [17], [S13]

Development languages

The GHC is developed by some languages

compiler
($(TOP)/compiler/*)

Haskell
+
Alex (lex)
Happy (yacc)
Cmm (C--)
Assembly

runtime system
($(TOP)/rts/*)

C
+
Cmm
Assembly

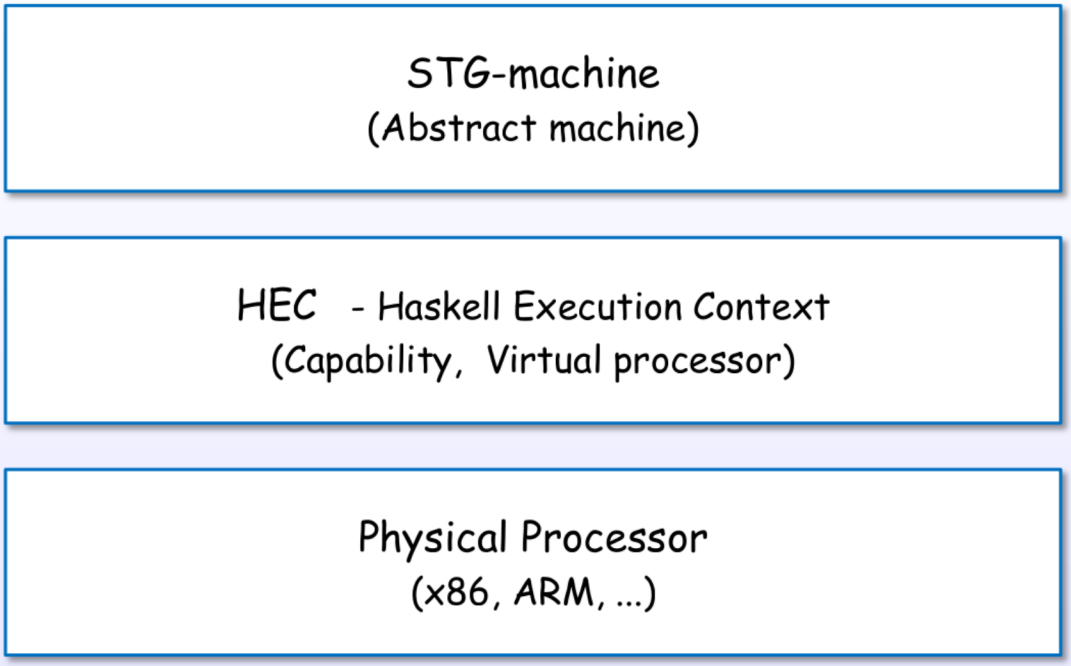
library
($(TOP)/libraries/*)

Haskell
+
C

References : [C2]

Machine layer/models

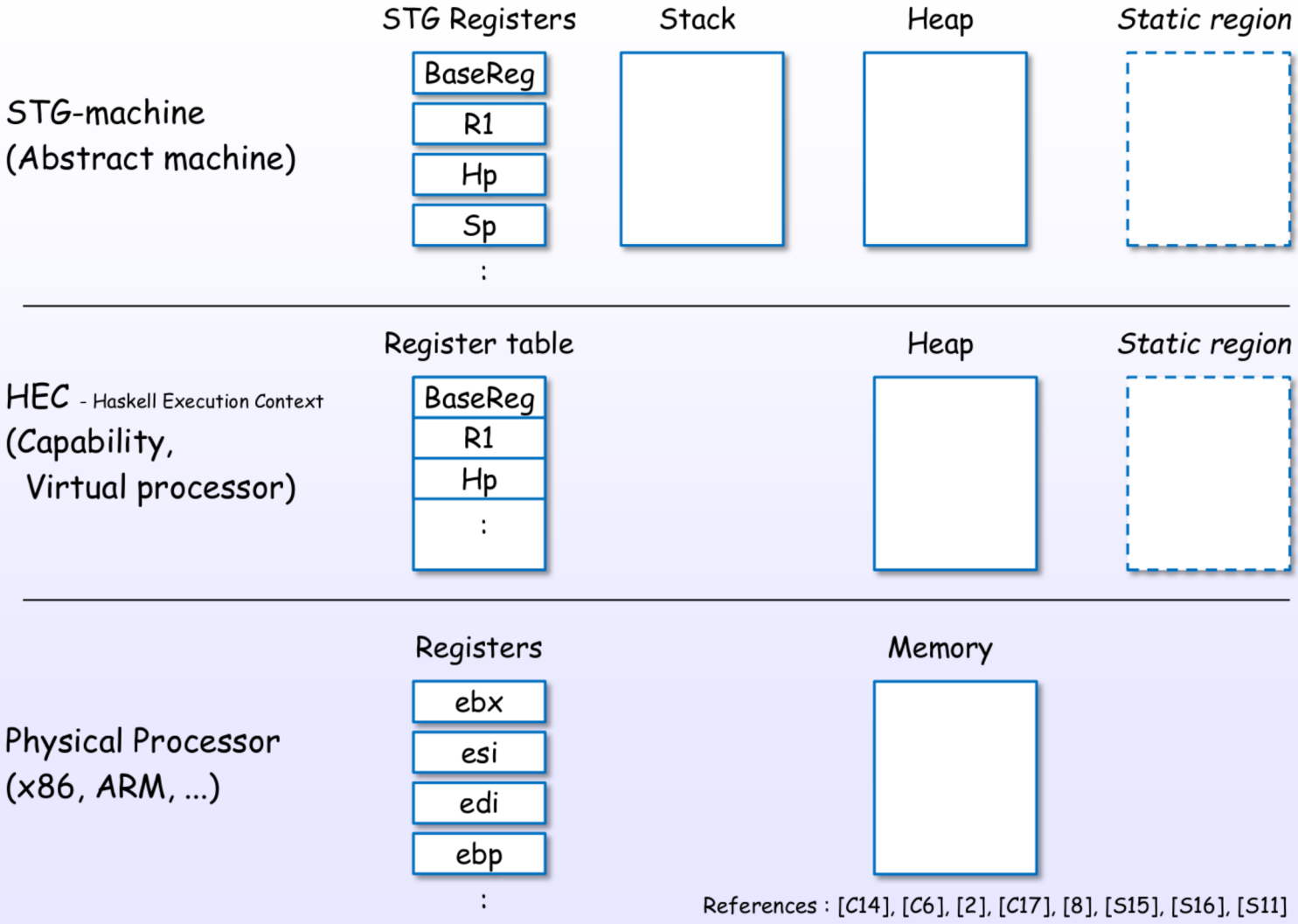
Machine layer



Each Haskell code is executed in STG semantics.

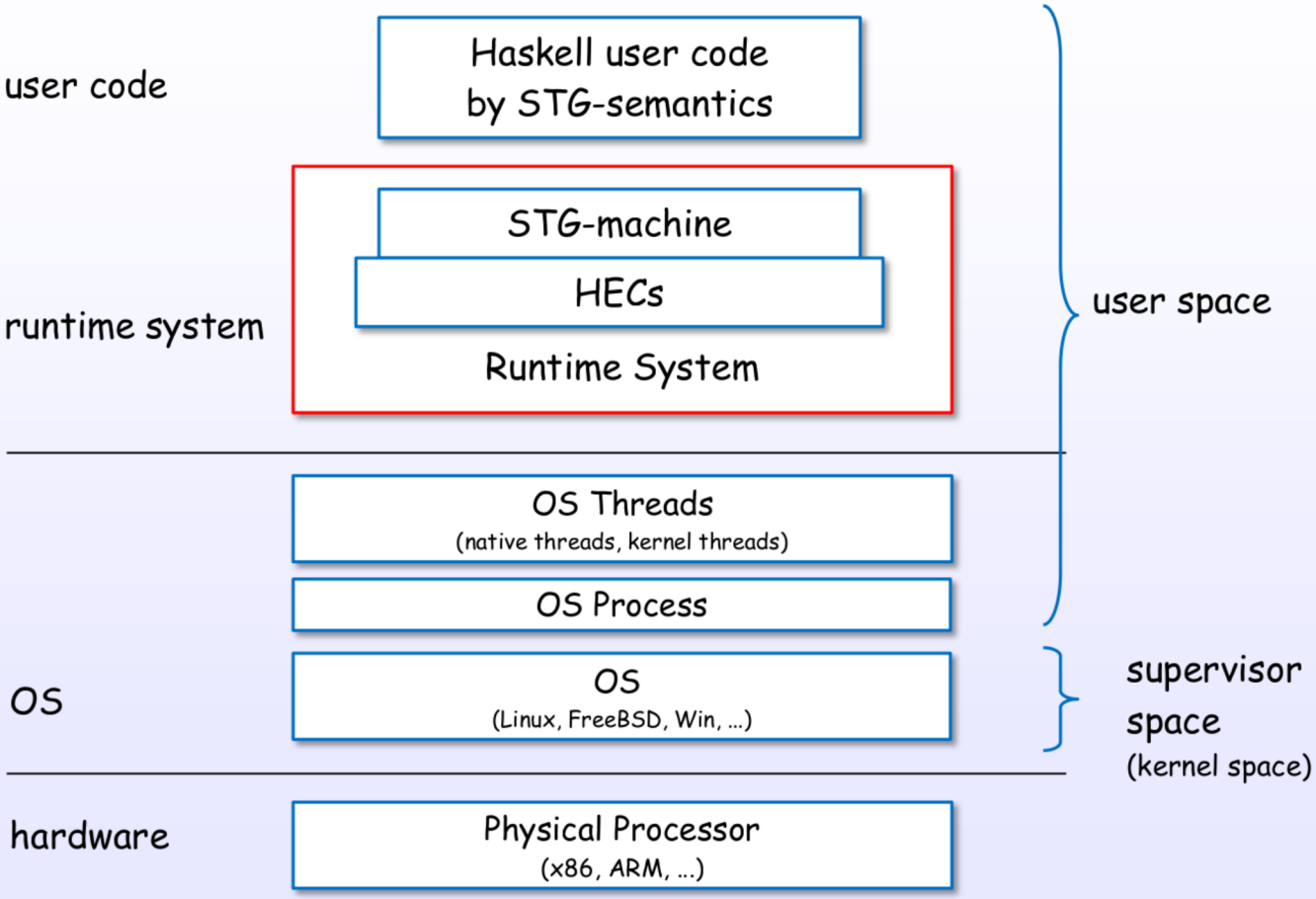
References : [C14], [C6], [2], [C17], [8], [S15], [S16], [S11]

Machine layer



References : [C14], [C6], [2], [C17], [8], [S15], [S16], [S11]

Runtime system and HEC

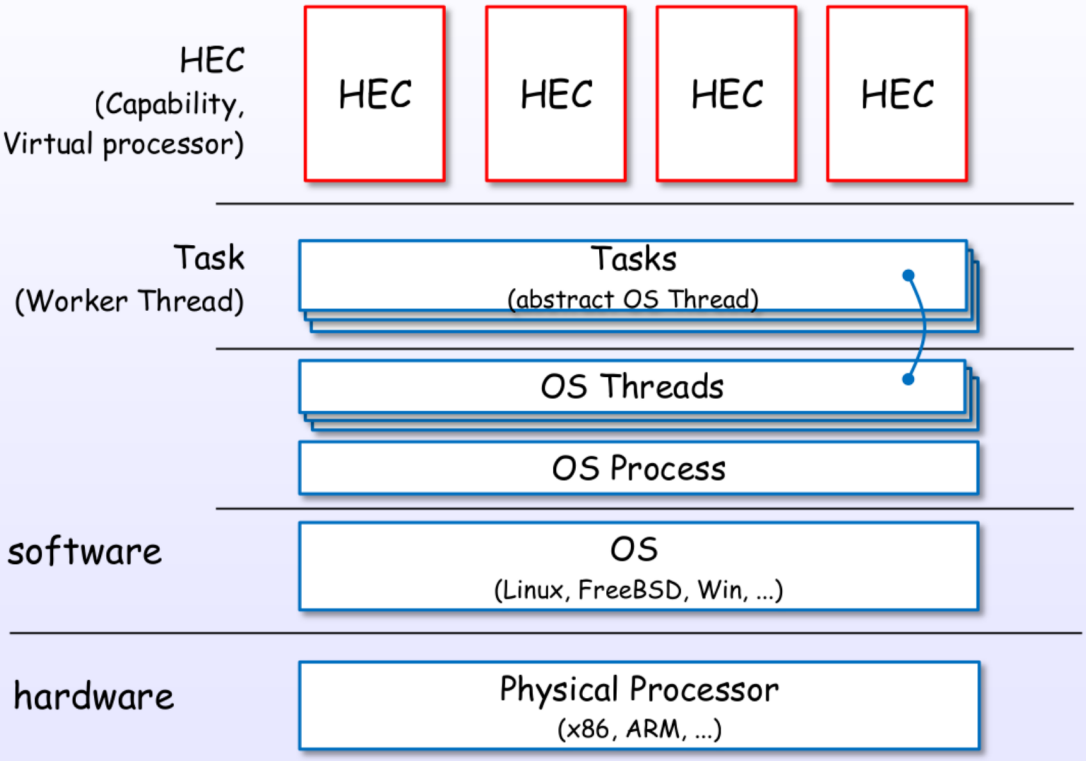


References : [C14], [C6], [2], [C17], [8], [S15], [S16], [S11]

many HECs

Multi HECs can be generated by compile and runtime options :

```
$ ghc -rtsopts -threaded
$ ./xxx +RTS -N4
```



References : [1], [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14]

HEC (Capability) data structure

[rts/Capability.h] (ghc 7.8)

```
struct Capability_ {
    StgFunTable f;
    StgRegTable r;
    nat no;
    Task *running_task;
    rtsBool in_haskell;
    nat idle;
    rtsBool disabled;
    StgTSO *run_queue_hd;
    StgTSO *run_queue_tl;
    InCall *suspended_ccalls;
    bdescr **mut_lists;
    bdescr **saved_mut_lists;
    bdescr *pinned_object_block;
    bdescr *pinned_object_blocks;
    int context_switch;
    int interrupt;

#if defined(THREADED_RTS)
    Task *spare_workers;
    nat n_spare_workers;
    Mutex lock;
    Task *returning_tasks_hd;
    Task *returning_tasks_tl;
    Message *inbox;
    SparkPool *sparks;
    SparkCounters spark_stats;
#endif

    W_ total_allocated;
    StgTVarWatchQueue *free_tvar_watch_queues;
    StgInvariantCheckQueue *free_invariant_check_queues;
    StgTRecChunk *free_trec_chunks;
    StgTRecHeader *free_trec_headers;
    nat transaction_tokens;
}
```

register table

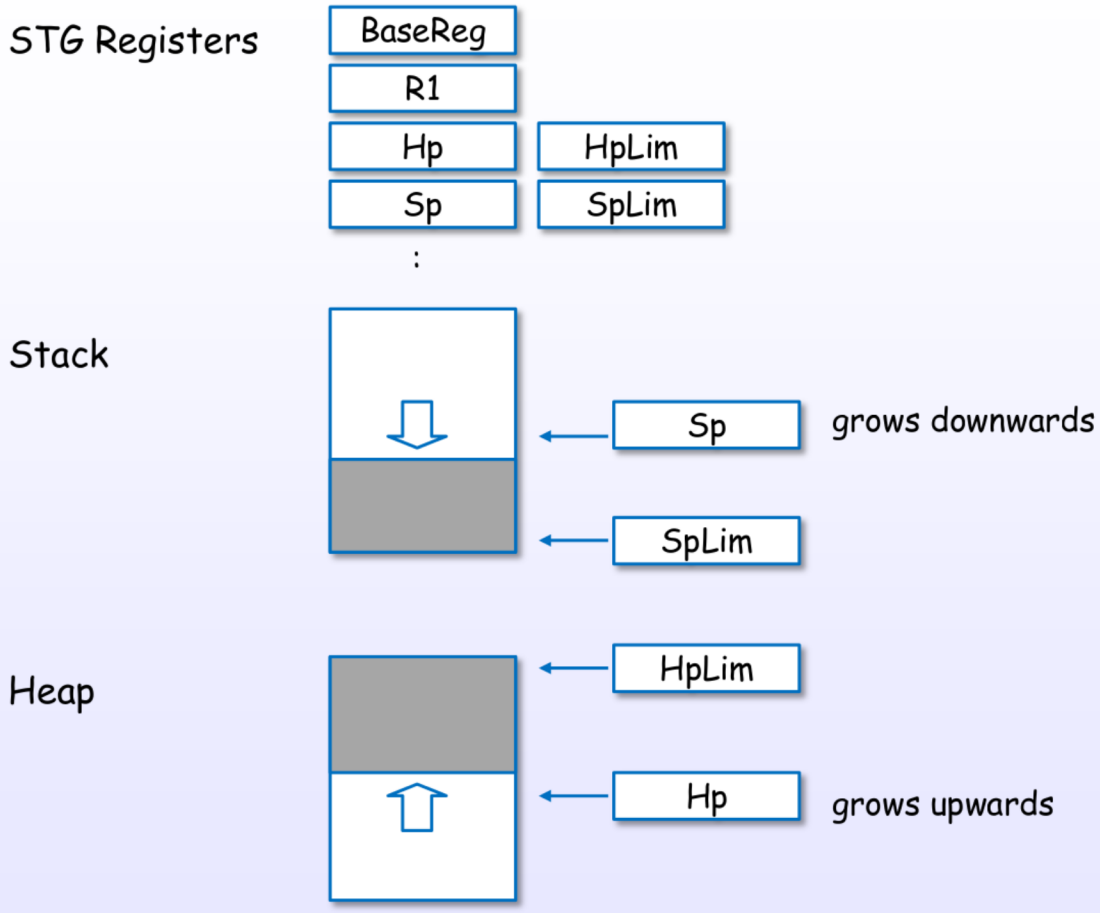
run queue

Each HEC (Capability) has a register table and a run queue and ...
Each HEC (Capability) is initialized at initCapabilities [rts/Capability.c]

References : [S15], [S16], [C11], [C17]

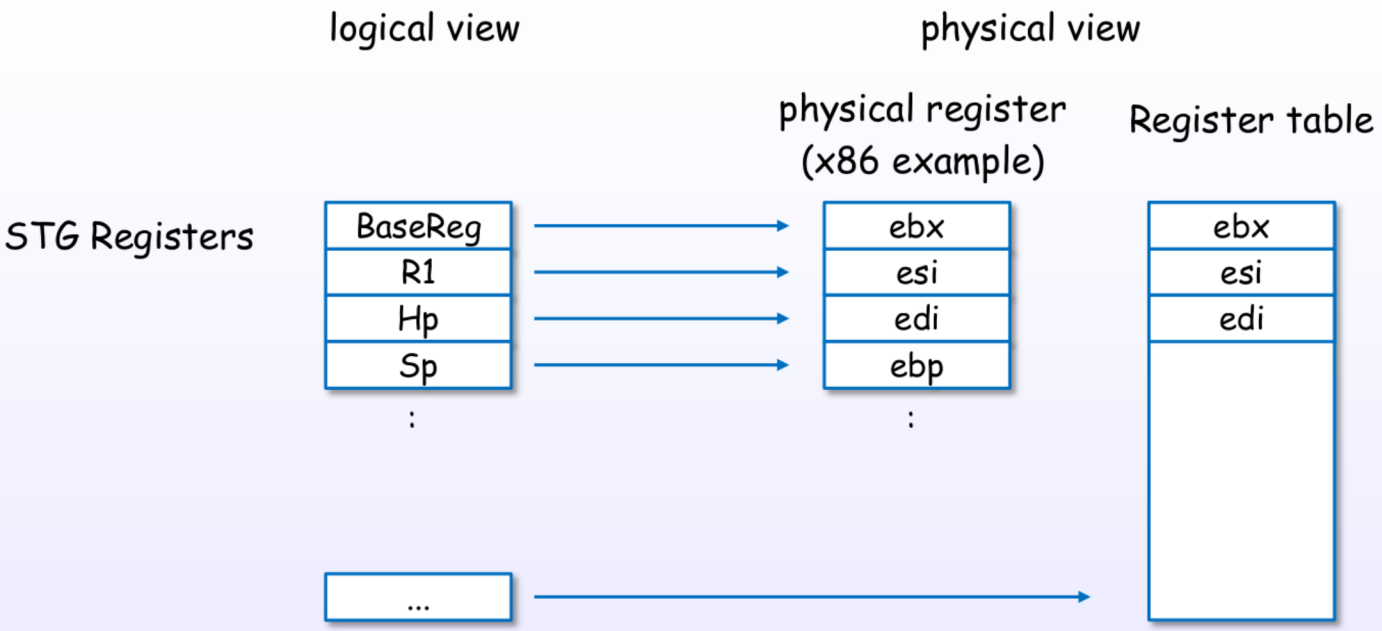
STG-machine

The STG-machine consists of three parts



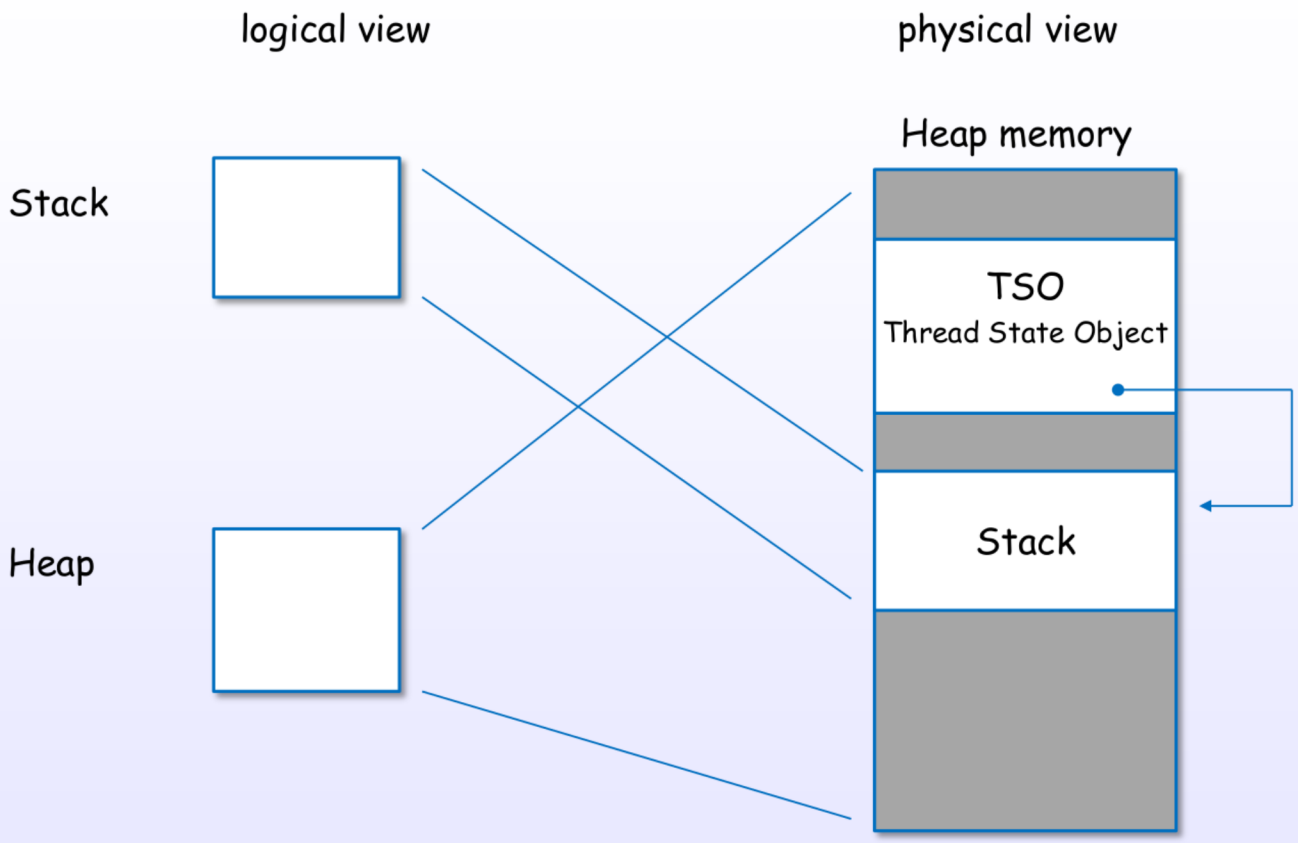
References : [2], [C15], [C11], [C12]

STG-machine is mapped to physical processor



References : [C15], [S1], [S2]

STG-machine is mapped to physical processor



A stack and a TSO object are in the heap.
The stack is stored separately from the TSO for size extension and GC.

References : [C11], [C12], [S16], [S5]

TSO data structure

[includes/rts/storage/TSO.h] (ghc 7.8)

```
typedef struct StgTSO_ {
   StgHeader            header;
   struct StgTSO_*       _link;
   struct StgTSO_*       global_link;
   struct StgStack_      *stackobj;
   StgWord16            what_next;
   StgWord16            why_blocked;
   StgWord32            flags;
   StgTSOBlockInfo       block_info;
   StgThreadID          id;
   StgWord32            saved_errno;
   StgWord32            dirty;
   struct InCall_*       bound;
   struct Capability_*    cap;
   struct StgTRecHeader_ * trec;
   struct MessageThrowTo_ * blocked_exceptions;
   struct StgBlockingQueue_ *bq;
   StgWord32  tot_stack_size;
} *StgTSOPtr;
```

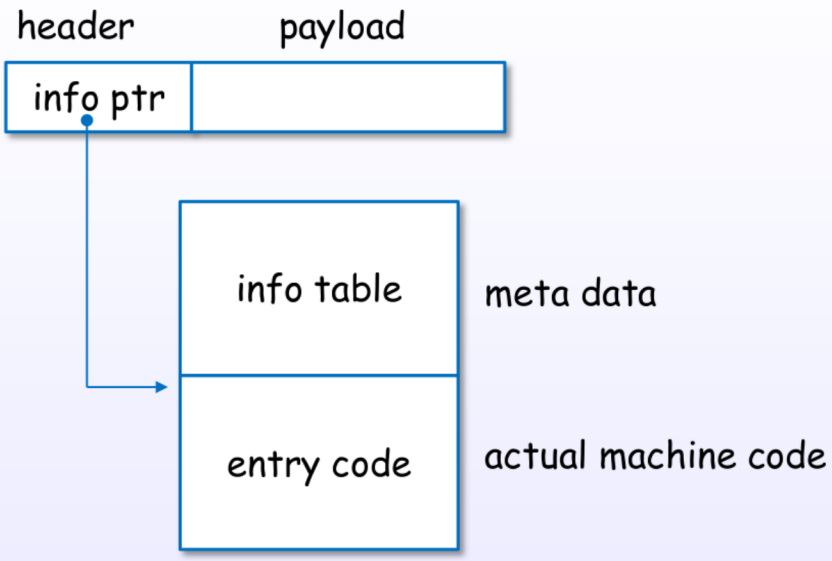
link to stack object

A TSO object is only ~17words + stack. Lightweight!

References : [S5]

Heap objects in STG-machine

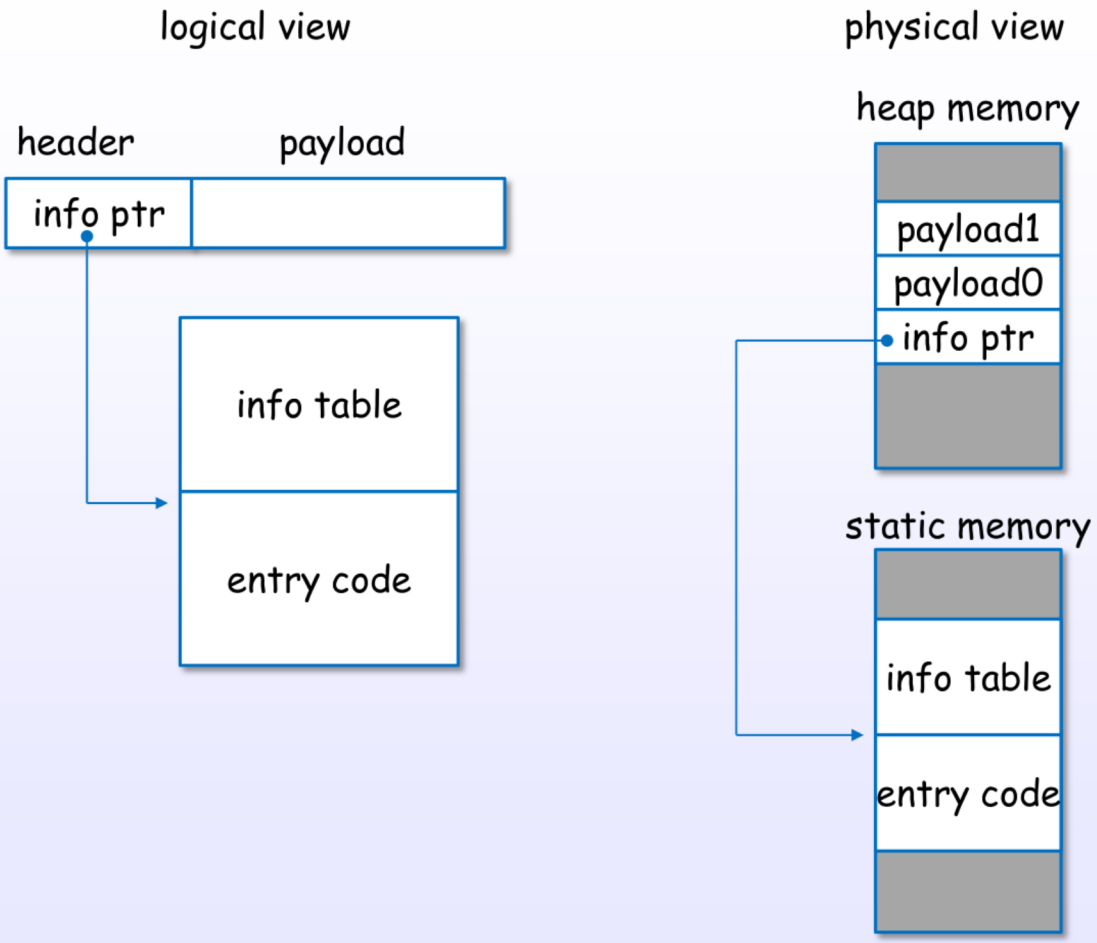
Every heap object is represented uniformly



Closure (header + payload) + Info Table + Entry Code

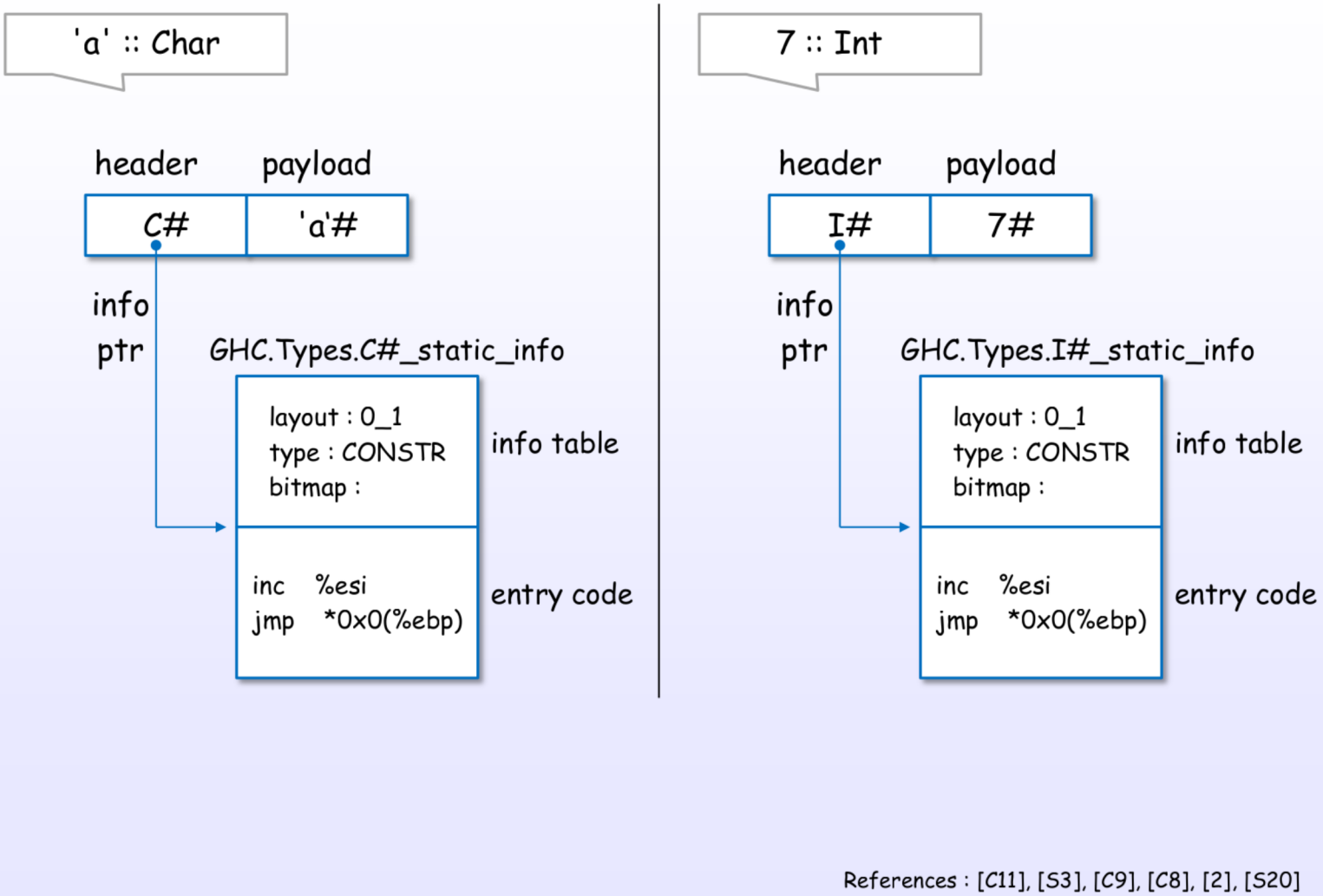
References : [C11], [S3], [S4], [S6], [2]

Heap object (closure)



References : [C11], [S3], [C9], [C8], [2]

Closure examples : Char, Int



References : [C11], [S3], [C9], [C8], [2], [S20]

Closure example (code)

[Example.hs]

```
module Example where
value1 :: Int
value1 = 7
```

STG

[ghc -O -ddump-stg Example.hs]

```
Example.value1 :: GHC.Types.Int
[GblId, Caf=NoCafRefs, Str=DmdType m, Unf=OtherCon []] =
    NO_CCS GHC.Types.I#! [8];
```

Cmm

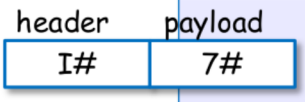
[ghc -O -ddump-opt-cmm Example.hs]

```
section "data" { __stginit_main:Example:
}

section "data" {
    Example.value1_closure:
        const GHC.Types.I#_static_info;
        const 7;
}

section "relreadonly" { SMc_srt:
}
```

asm

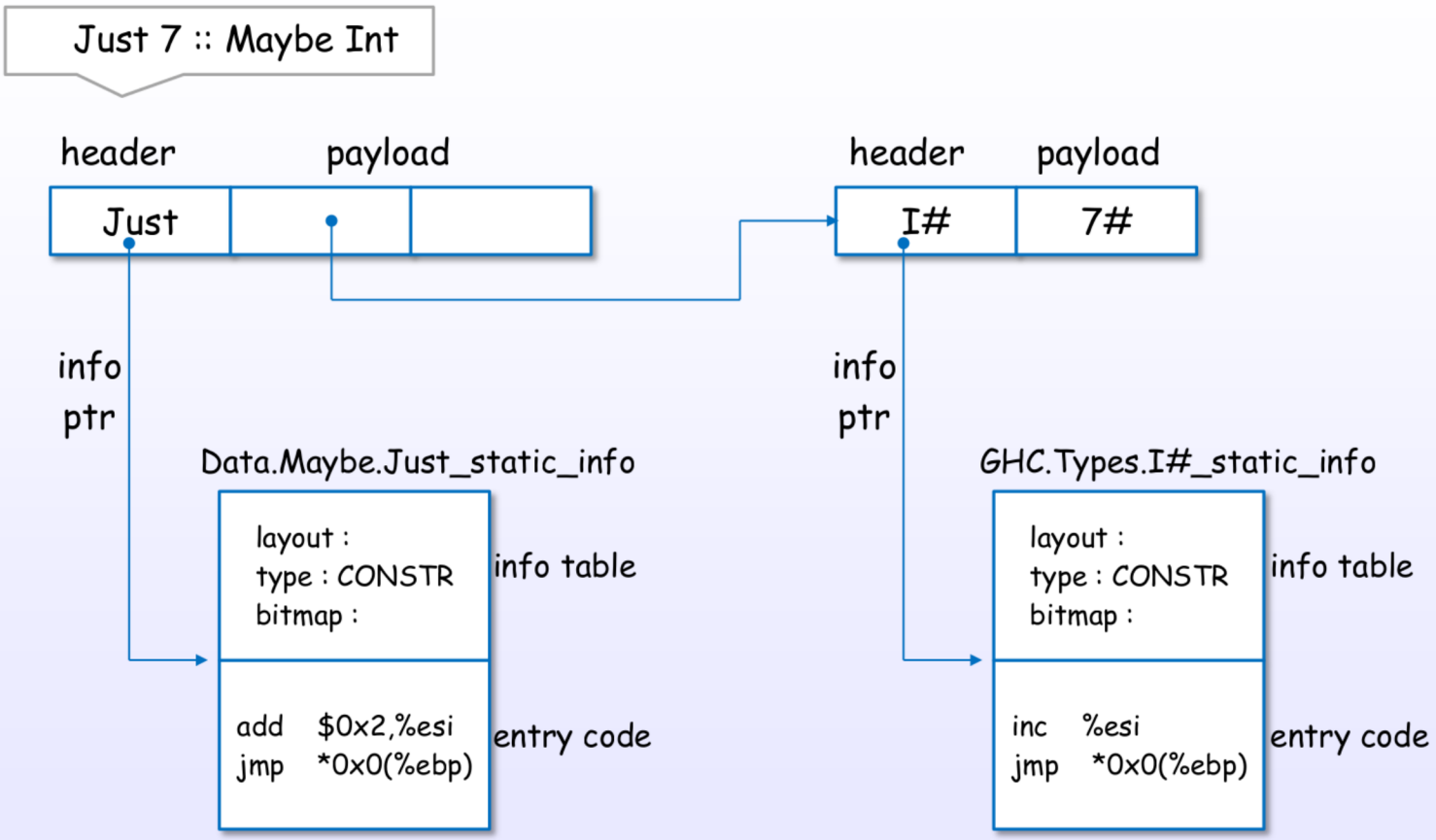
[ghc -O -ddump-asm Example.hs]

```
.data
        .align 4
.align 1
.globl __stginit_main:Example
__stginit_main:Example:
.data
        .align 4
.align 1
.globl Example.value1_closure
Example.value1_closure:
        .long   GHC.Types.I#_static_info
        long    7
.section .data
        .align 4
.align 1
SMd_srt:
```



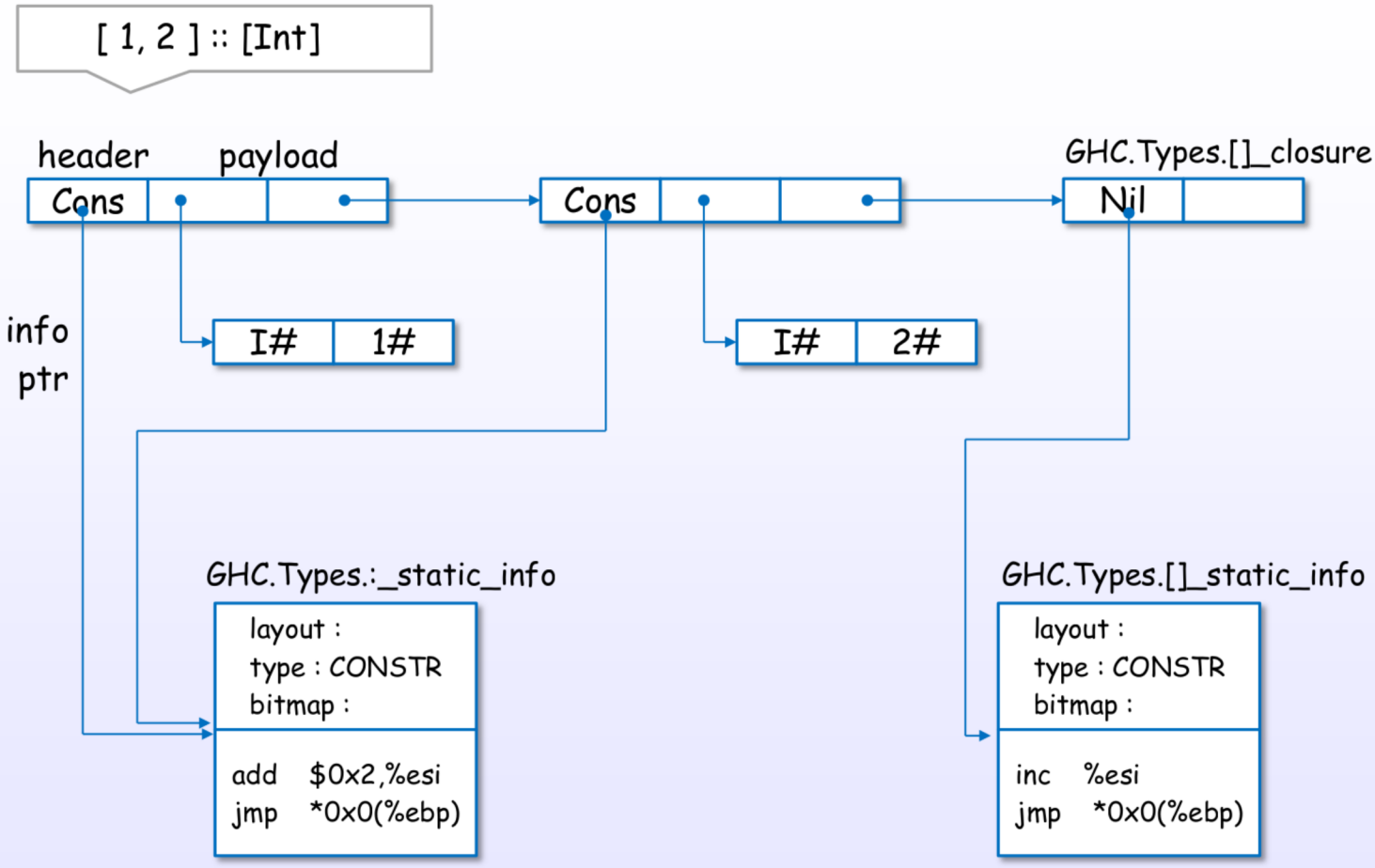
References : [C11], [S3], [C9], [C8], [2], [S20]

Closure examples : Maybe



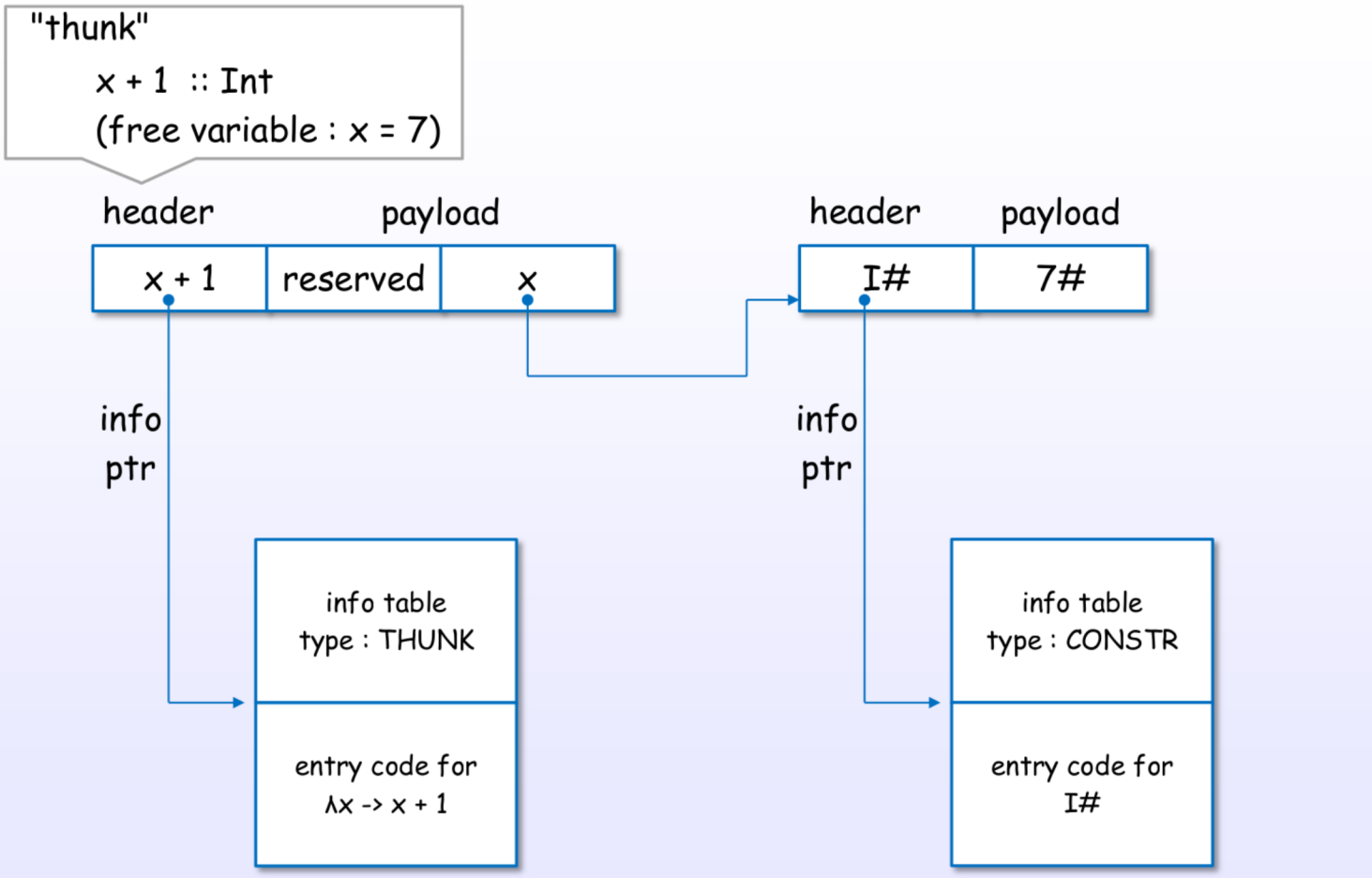
References : [C11], [S3], [C9], [C8], [2], [S20]

Closure examples : List



References : [C11], [S3], [C9], [C8], [2], [S20]

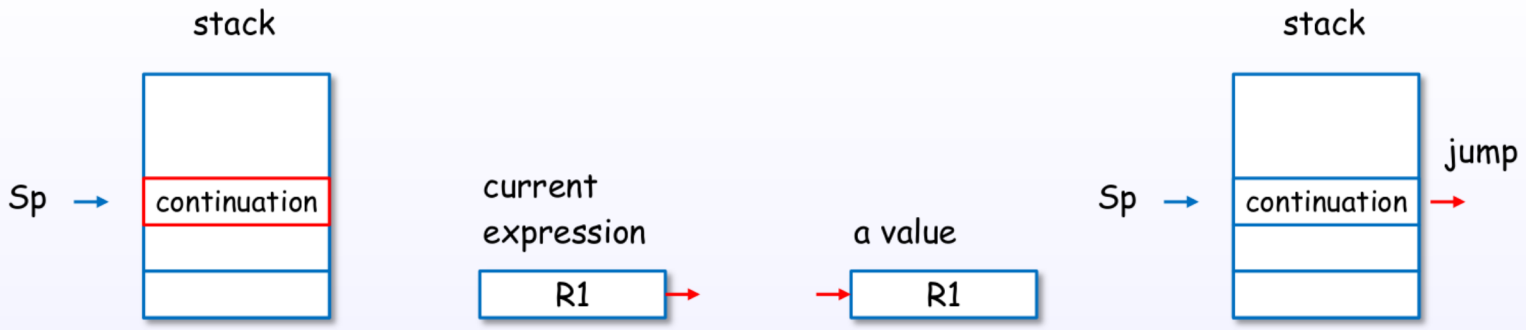
Closure examples : Thunk



References : [C11], [S3], [C9], [C8], [2], [S20]

STG-machine evaluation

STG evaluation flow



(1)
push a continuation code (next code) to the stack top

(2)
enter to R1 closure

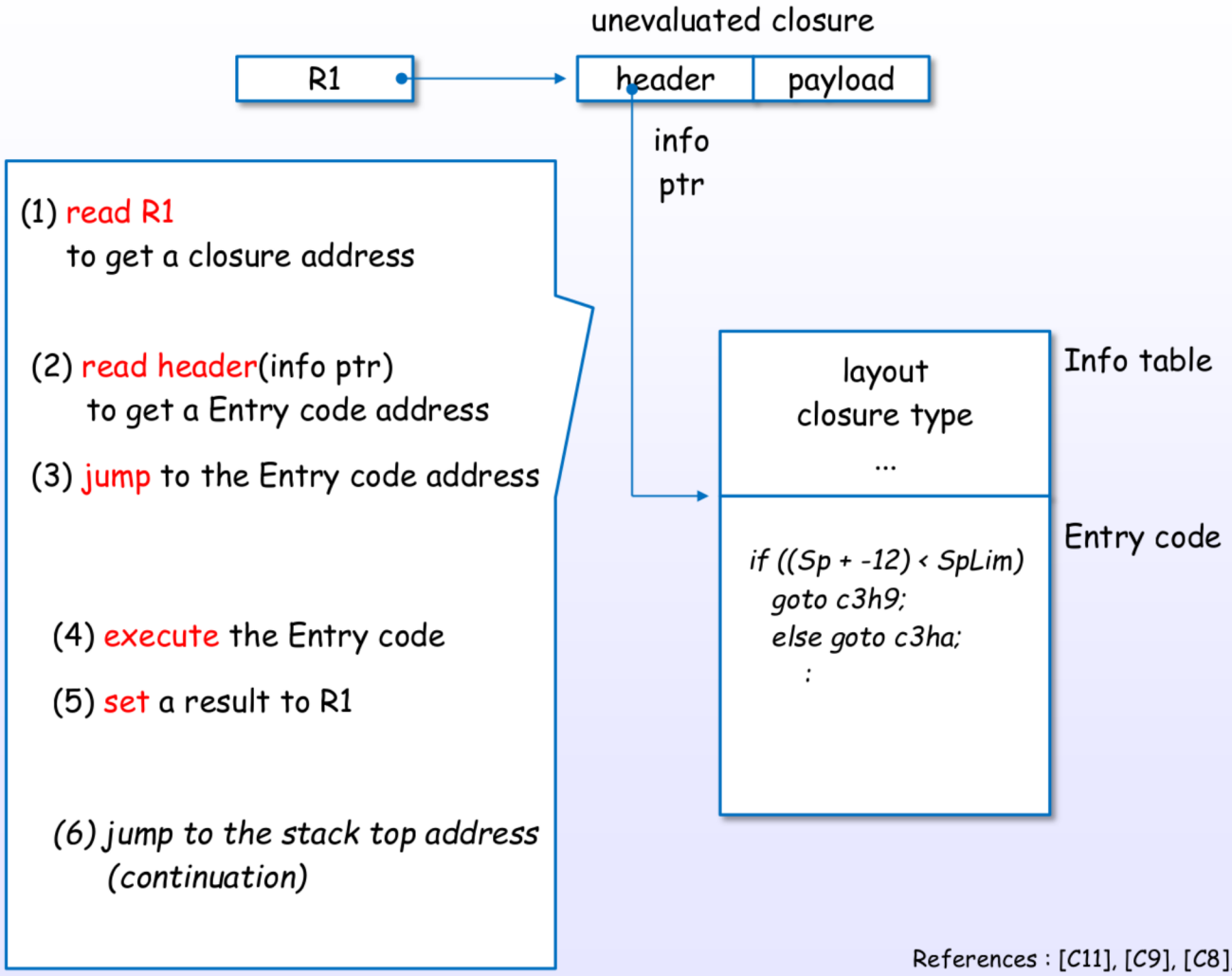
(3)
set a result to R1

(4)
jump (return) to the stack top code

(5)
repeat from (1)

References : [C8], [3]

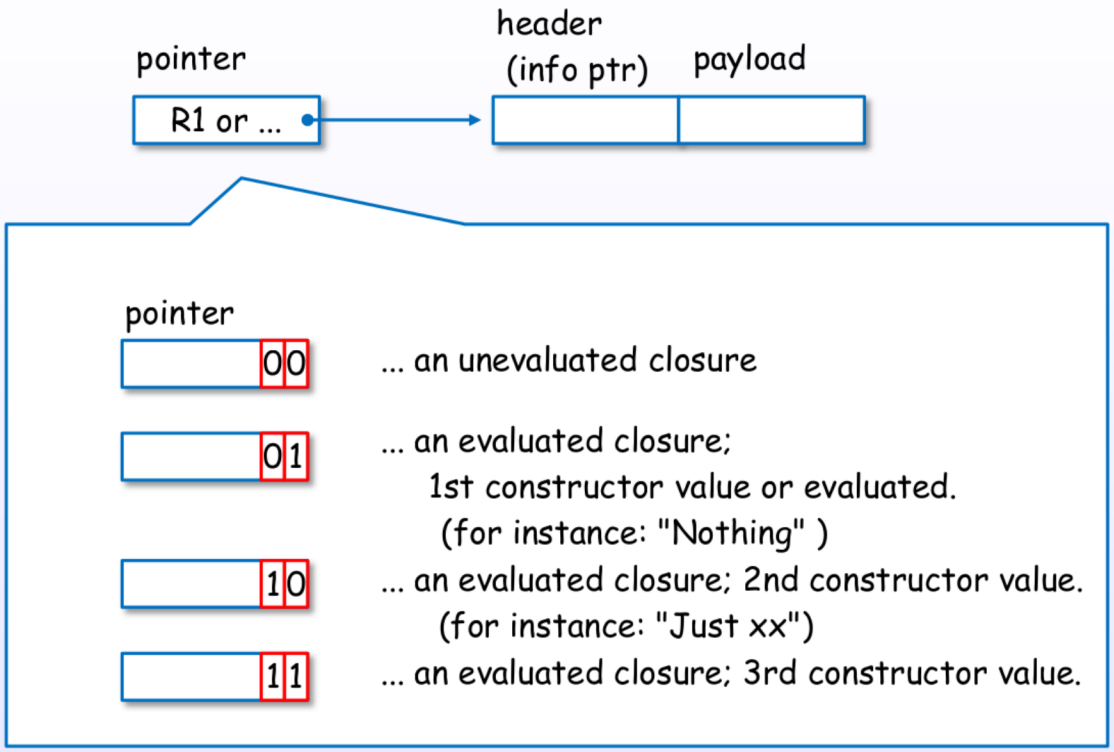
Enter to a closure



References : [C11], [C9], [C8], [10], [3], [2], [12]

Pointer tagging

Pointer tagging



* 32bit machine case

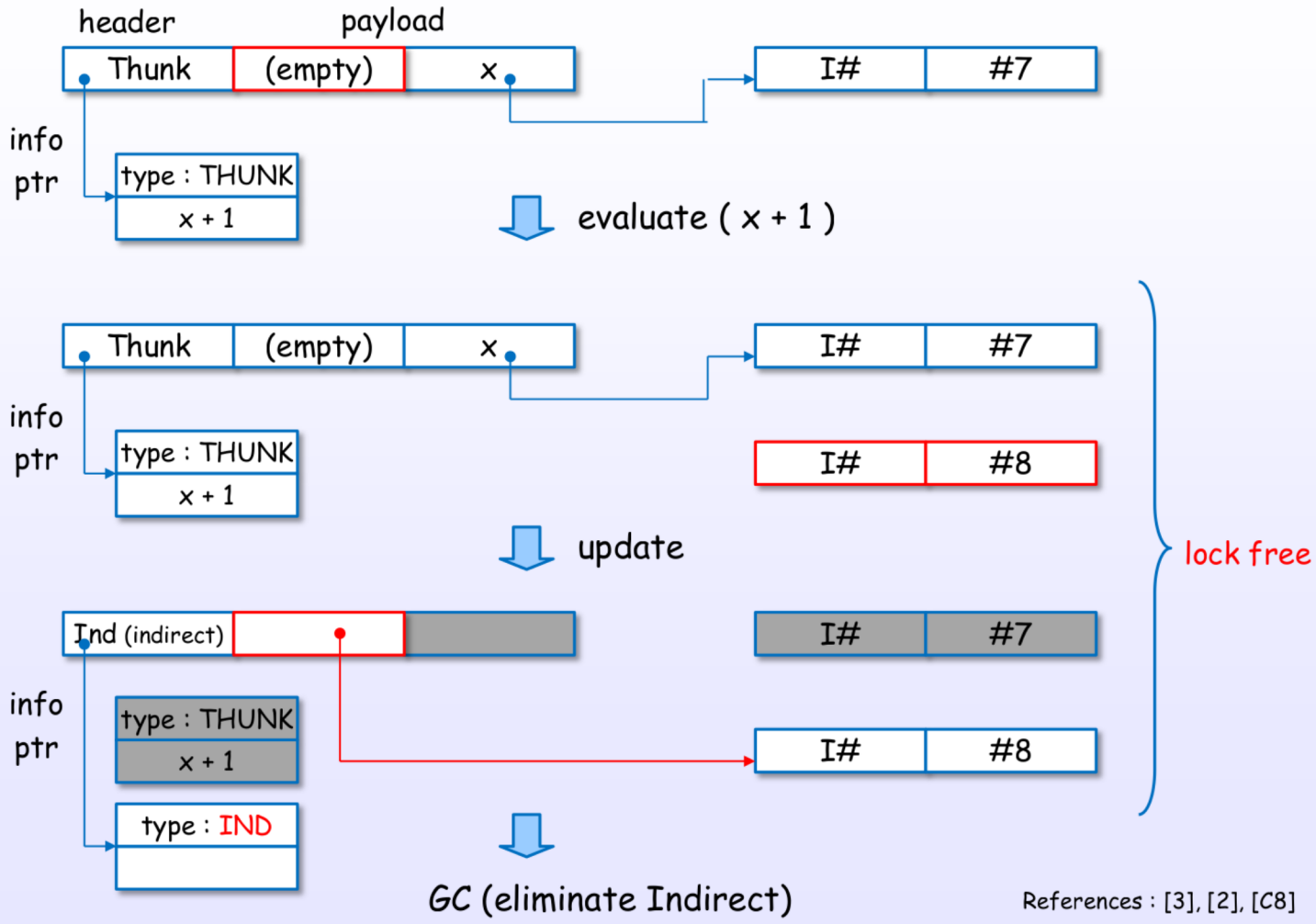
fast judgment!
check only pointer's lower bits without evaluating the closure.

References : [4], [2], [C16]

Thunk and update

Thunk and update

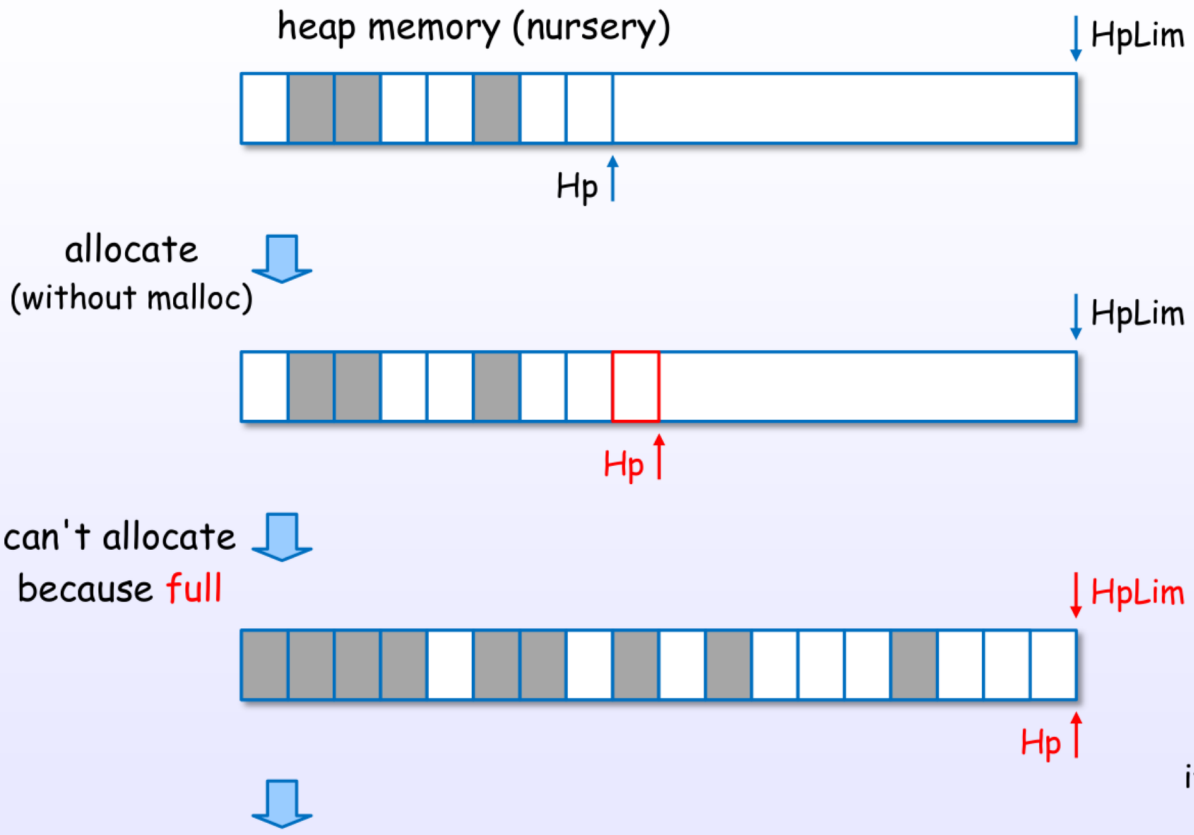
"thunk" x + 1 :: Int (free variable : x = 7)



References : [3], [2], [C8]

Allocate and free heap objects

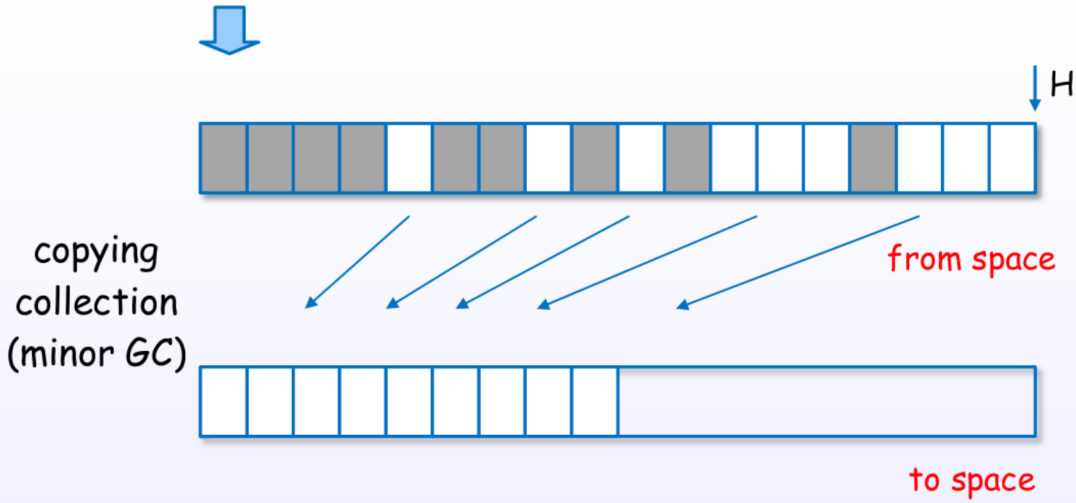
Allocate heap objects



if (Hp > HpLim) goto ...
call stg_gc_...

References : [C11], [C13], [8], [9], [5], [15], [12], [13], [19], [S25]

free and collect heap objects

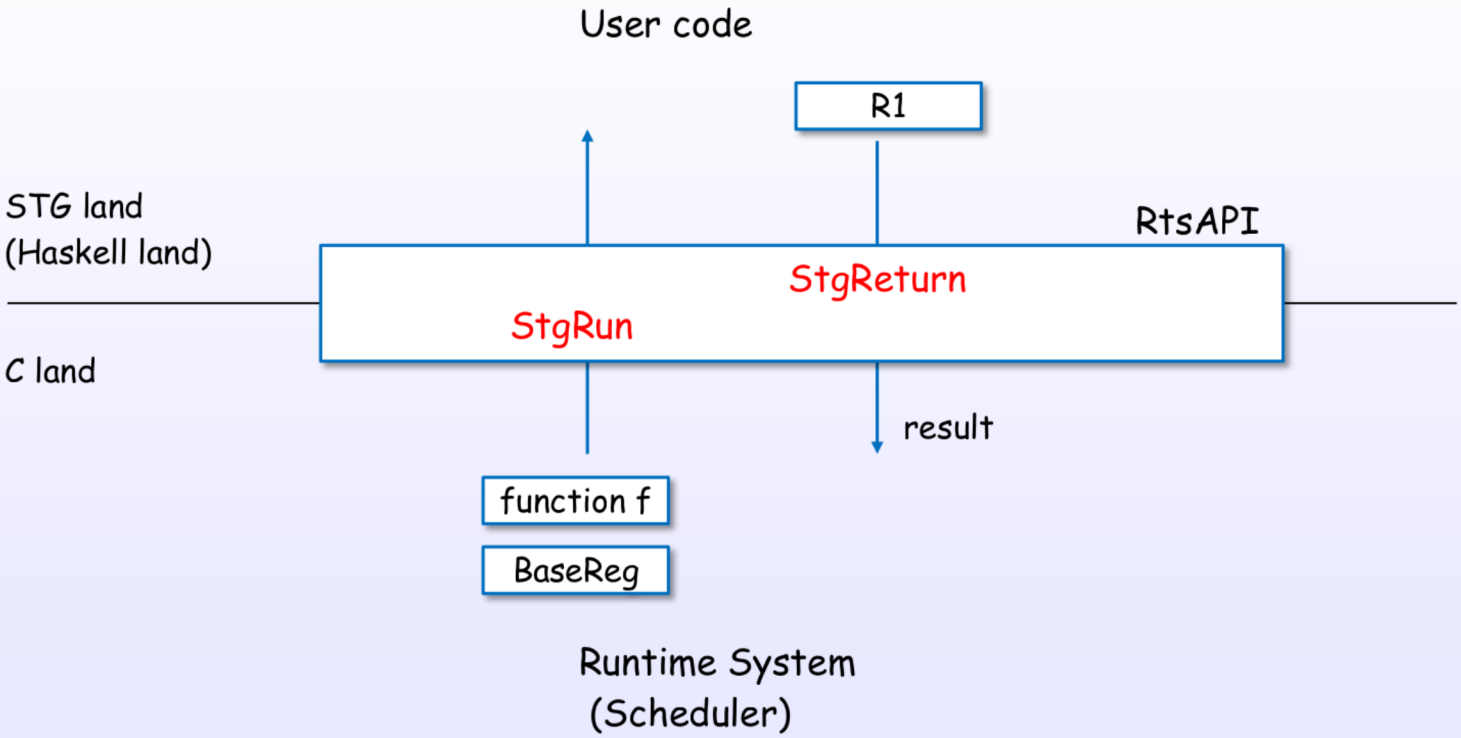


if (Hp > HpLim) goto ...
call stg_gc_...

References : [C11], [C13], [8], [9], [5], [15], [12], [13], [19], [S25]

STG - C land interface

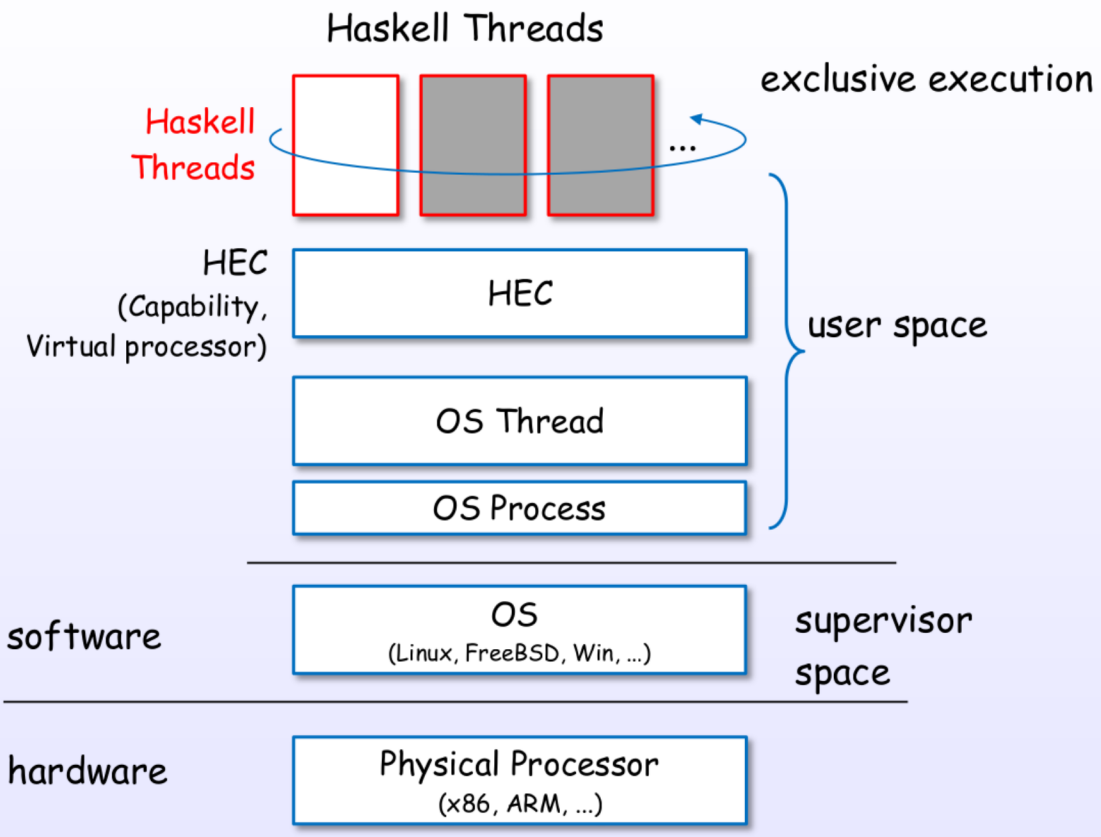
STG (Haskell) land - C land interface



References : [S18], [S17], [S19], [S21]

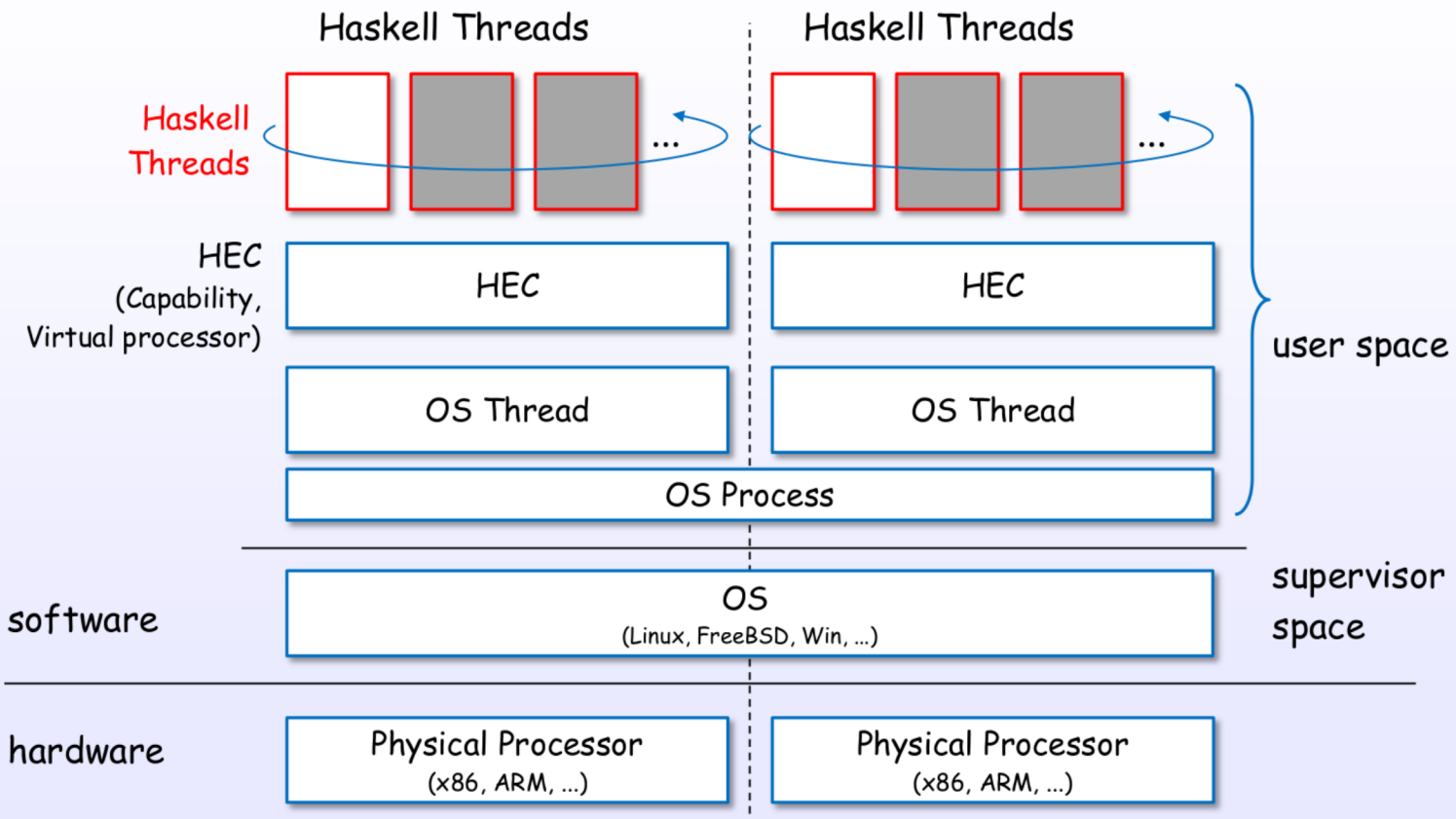
Thread

Thread layer (single core)



References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14]

Thread layer (multi core)

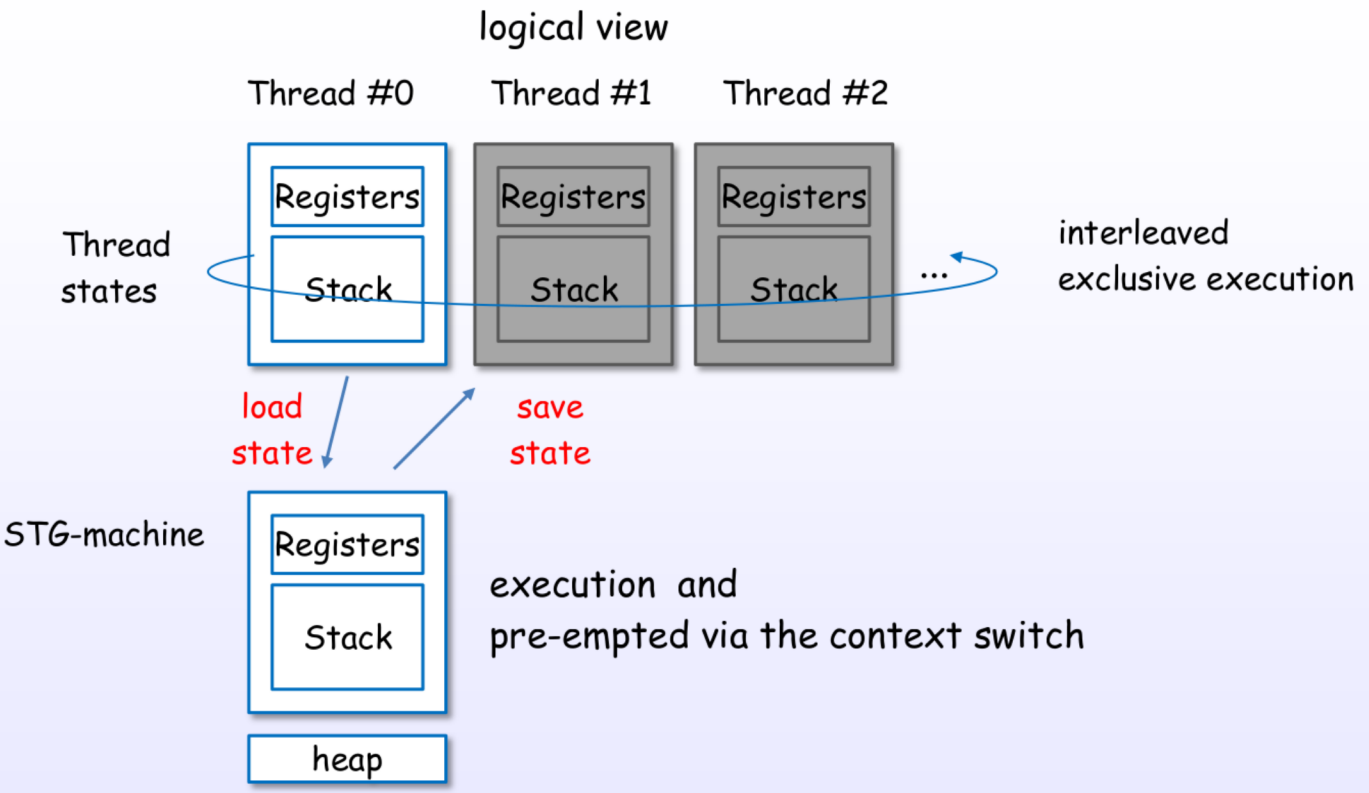


*Threaded option case (ghc -threaded)

References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14]

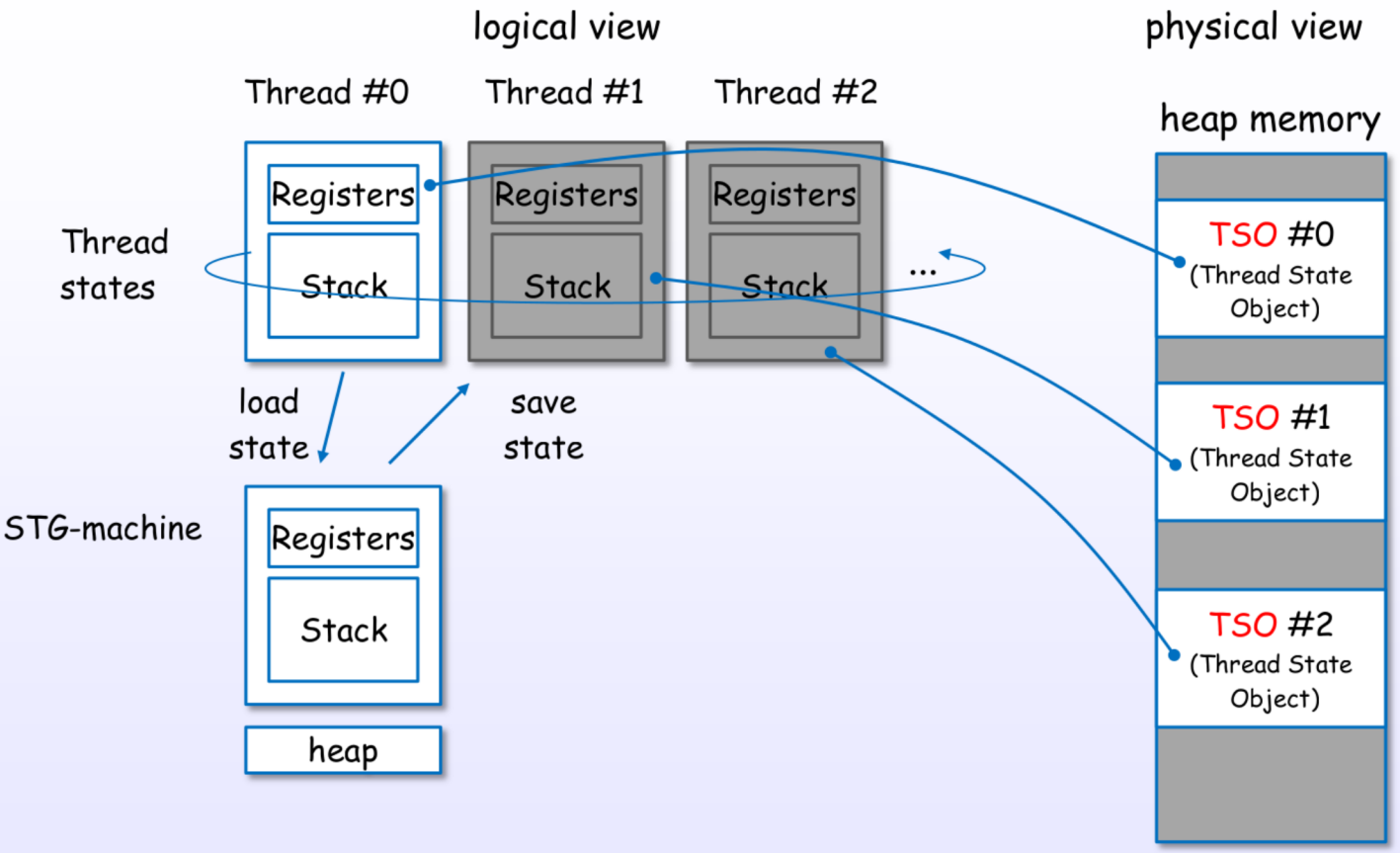
Thread context switch

Threads and context switch



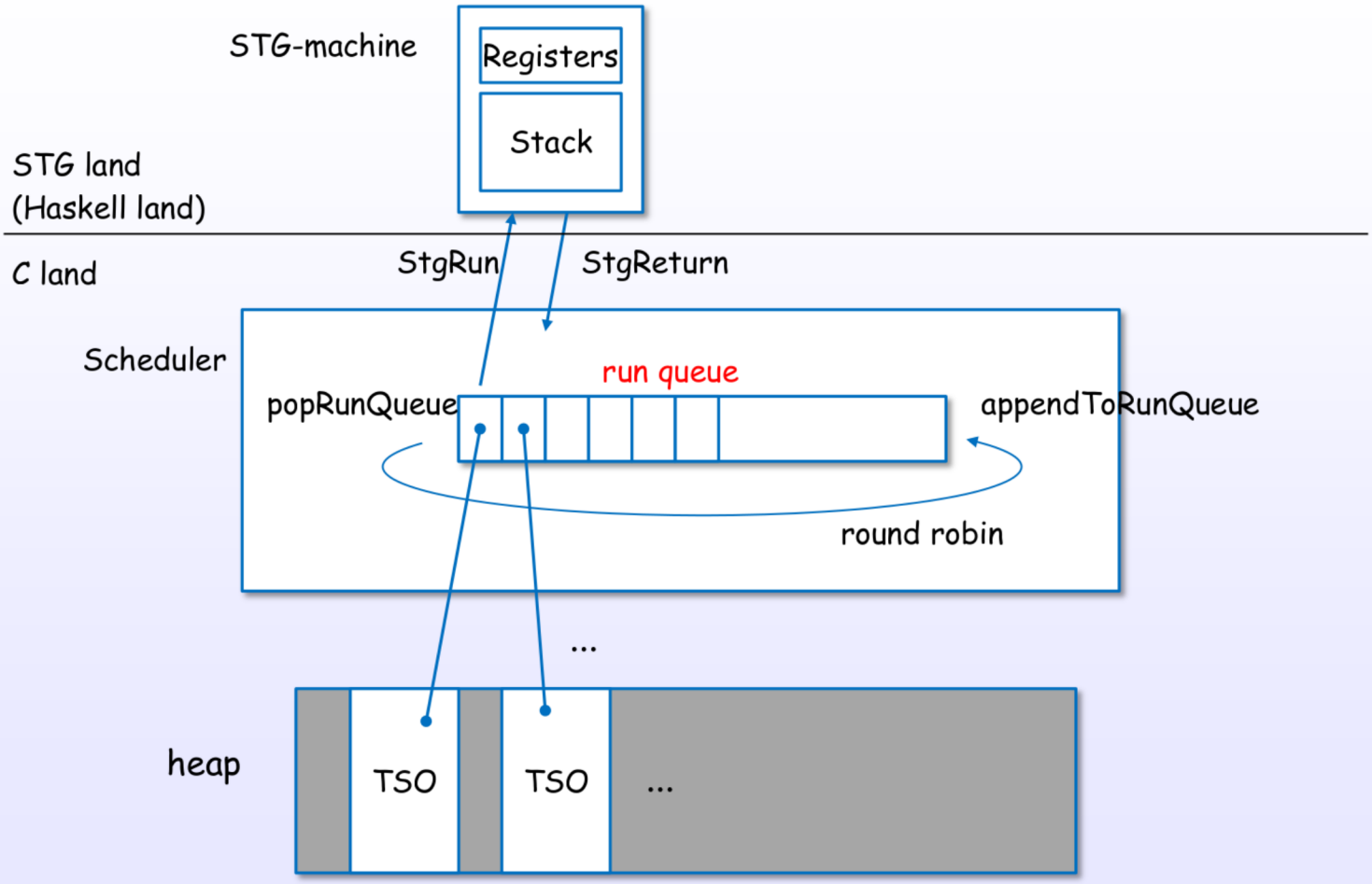
References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14]

Threads and TSOs



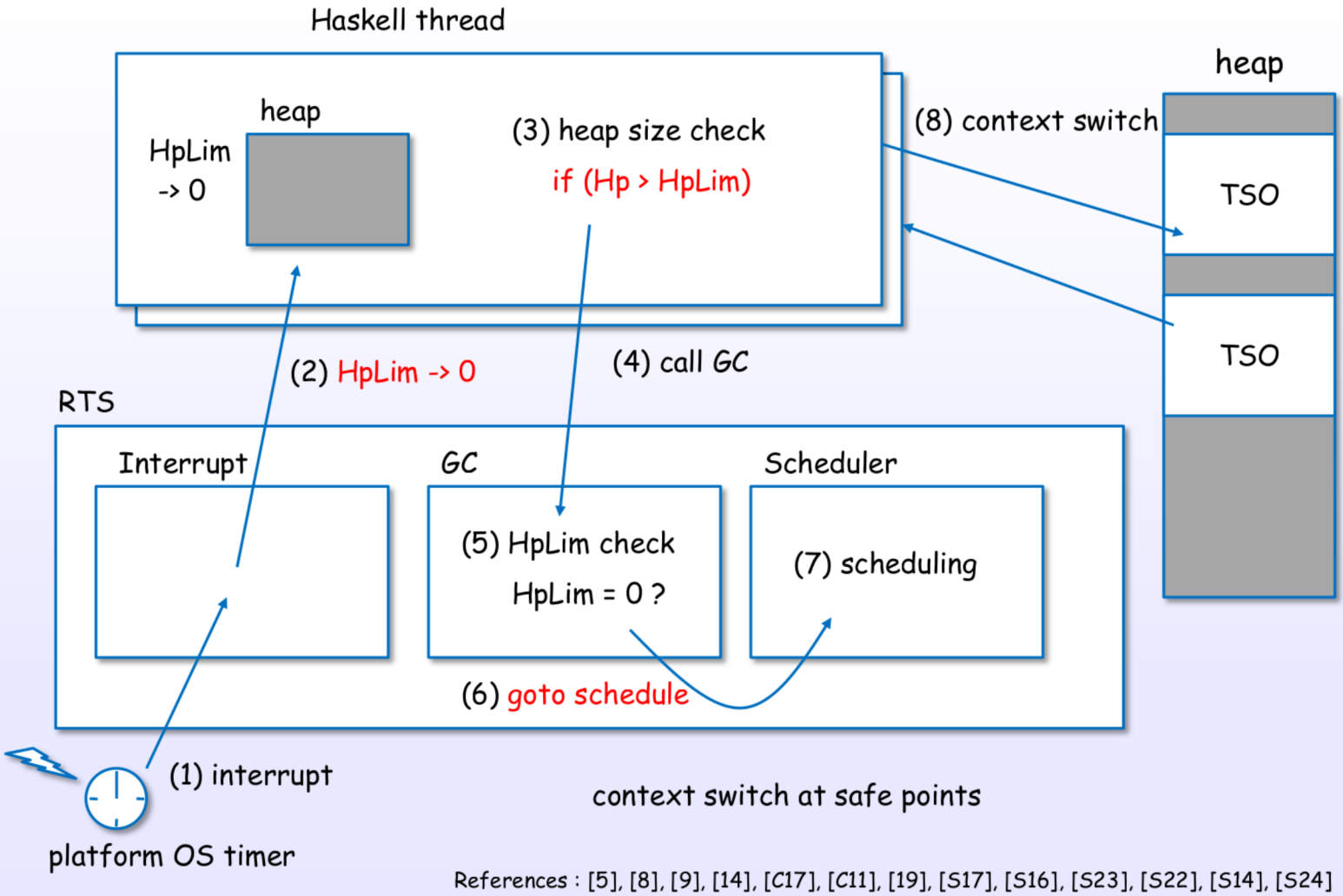
References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14]

Scheduling by run queue



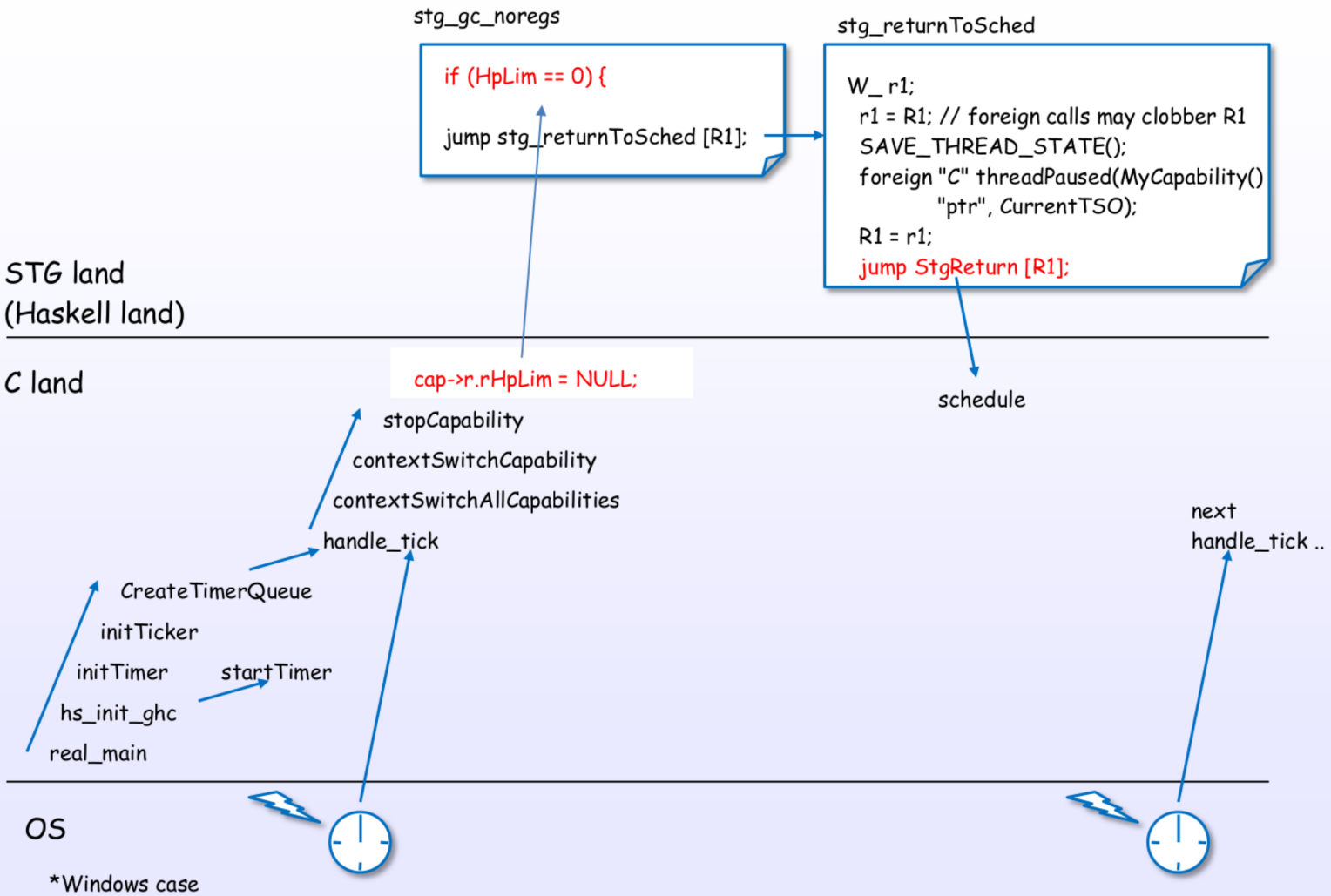
References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14]

Context switch flow



References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14], [S24]

Context switch flow (code)



References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14], [S24]

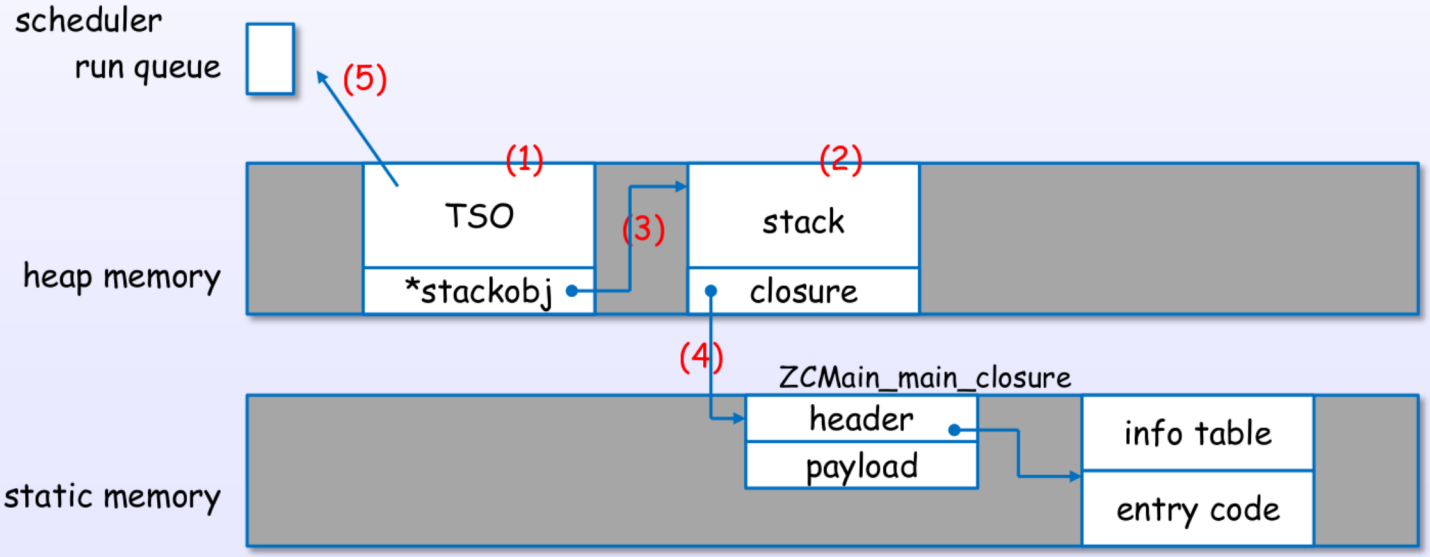
Creating main and sub threads

Create a main thread

Runtime System

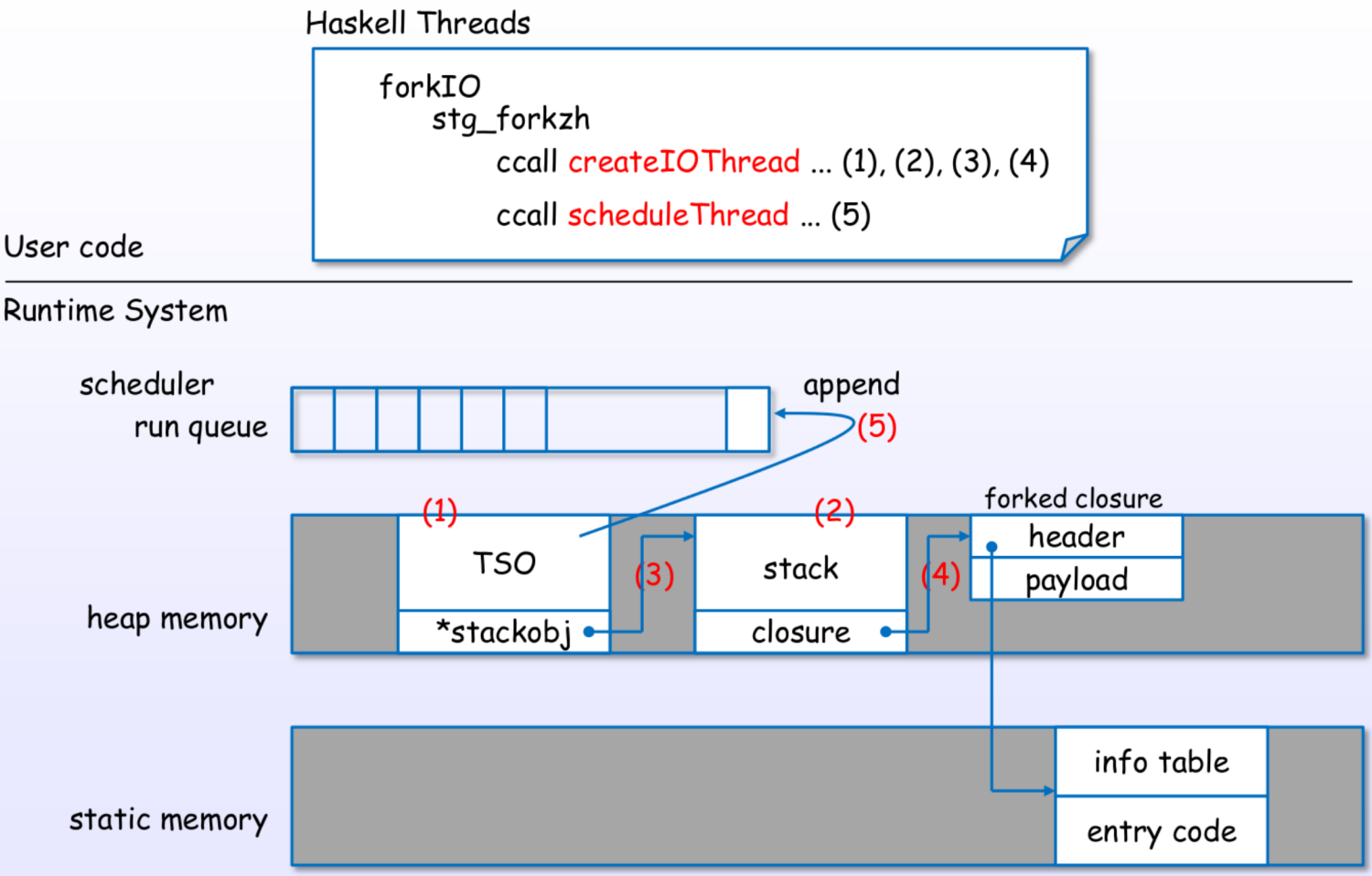
Runtime system bootstrap code [rts/RtsAPI.c]

```
rts_evalLazyIO
    createIOThread
        createThread ... (1), (2), (3)
        pushClosure  ... (4)
    scheduleWaitThread
        appendToRunQueue ... (5)
```



References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14], [S24]

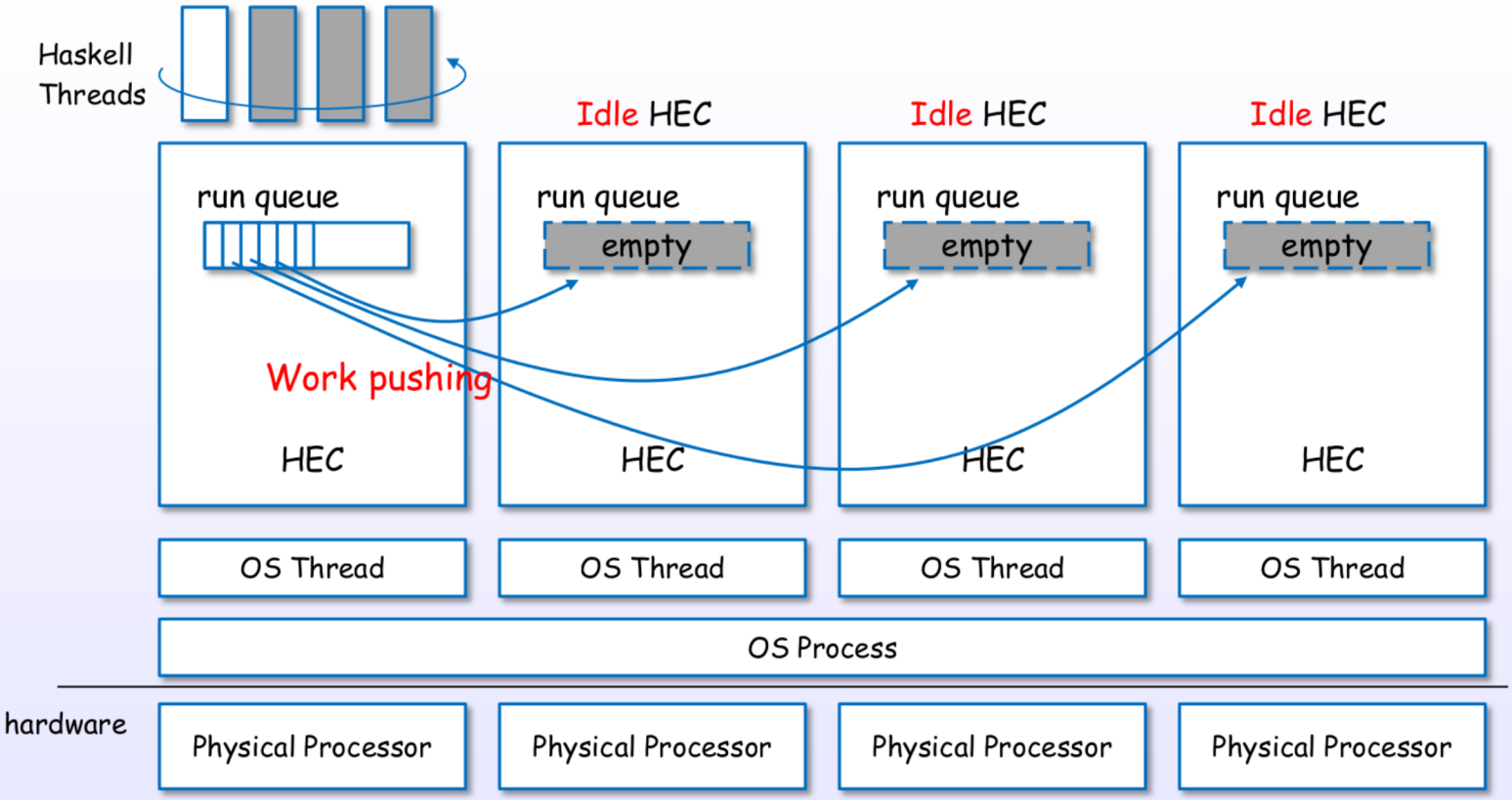
Create a sub thread using forkIO



References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14], [S24]

Thread migration

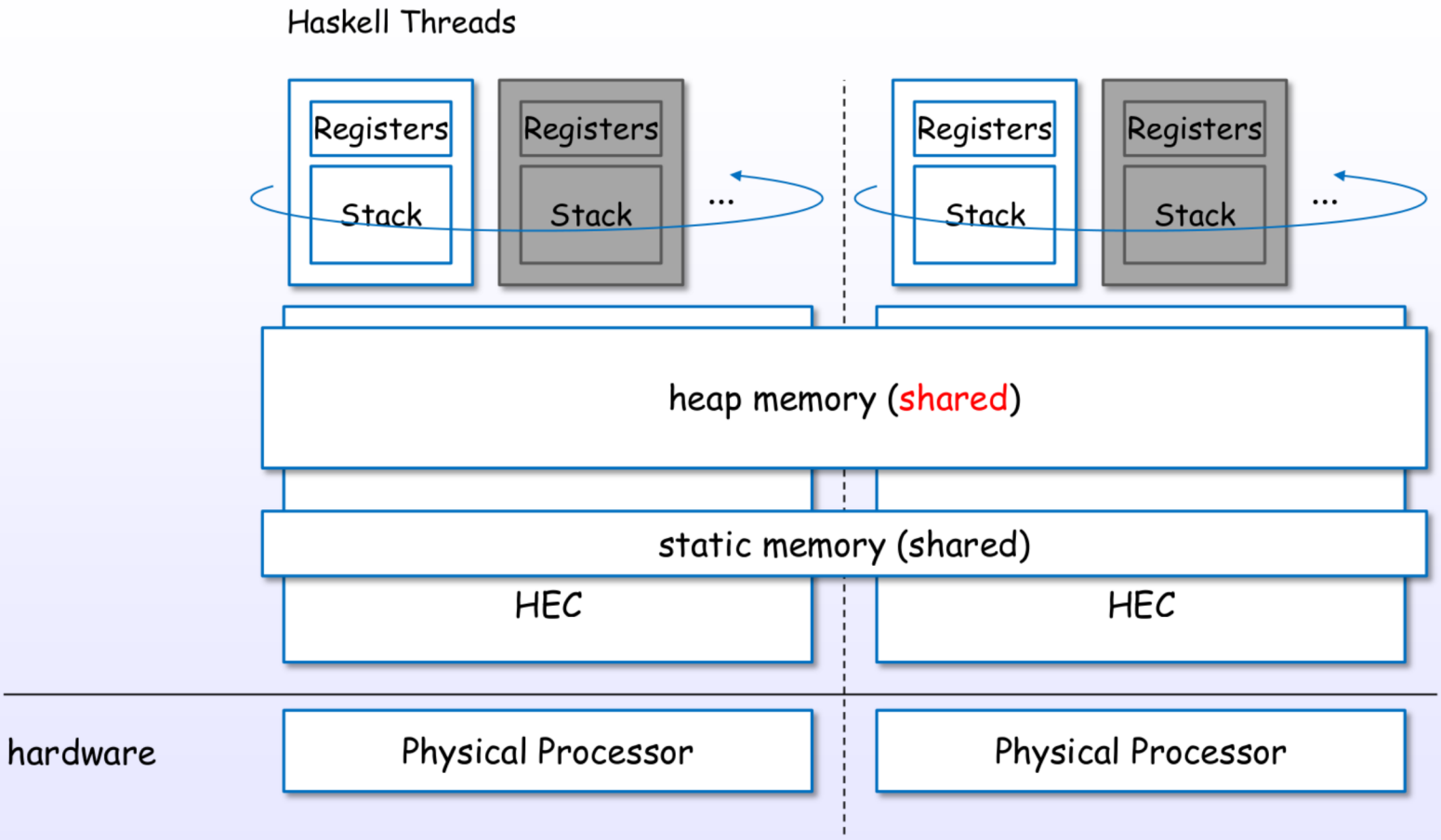
Threads are migrated to idle HECs



References : [5], [8], [9], [14], [C17], [C18], [S17], [S16], [S23], [S24]

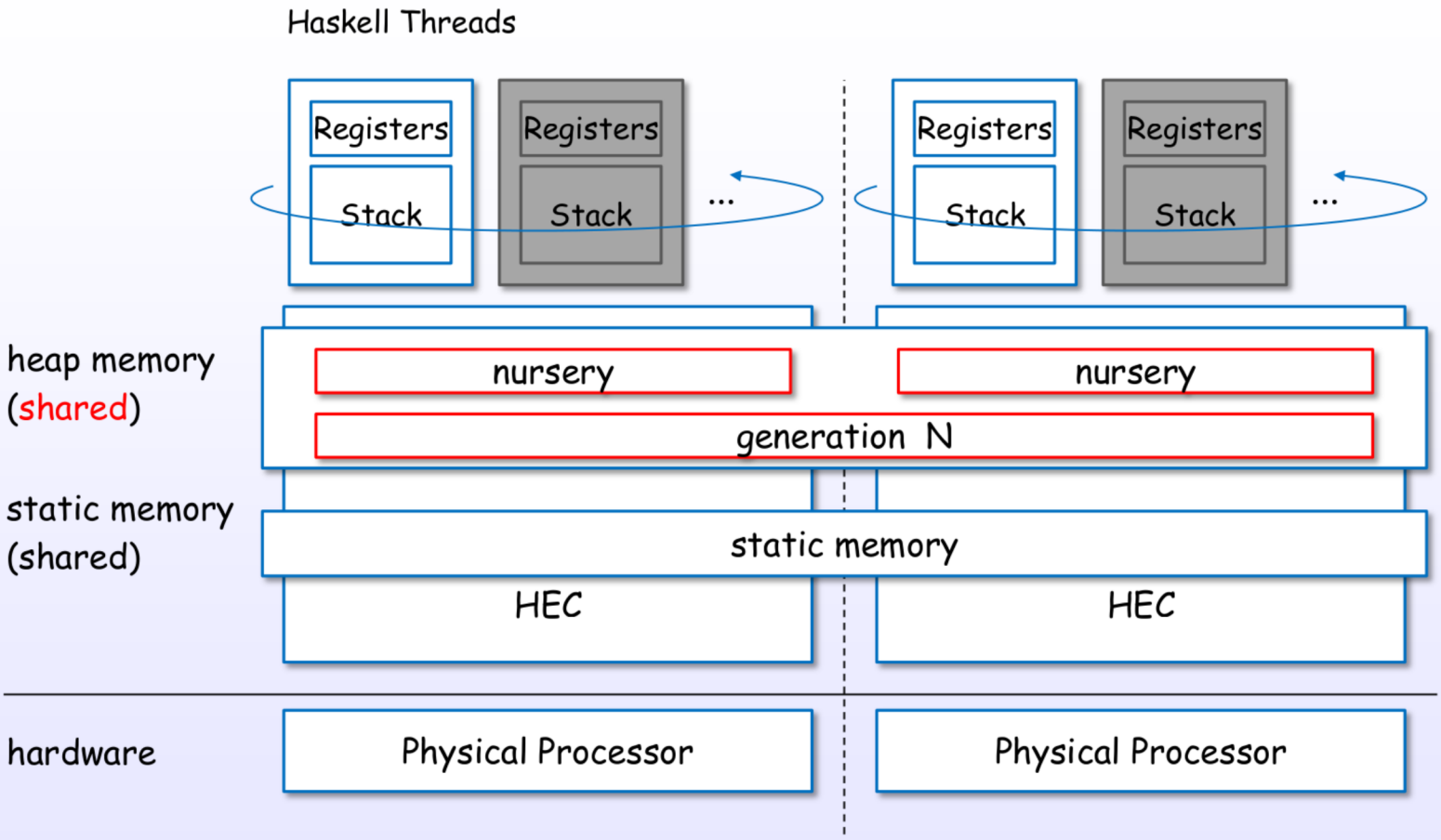
Heap and Threads

Threads shared a heap



References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14], [S17], [S16], [S25]

Local allocation area (nursery)

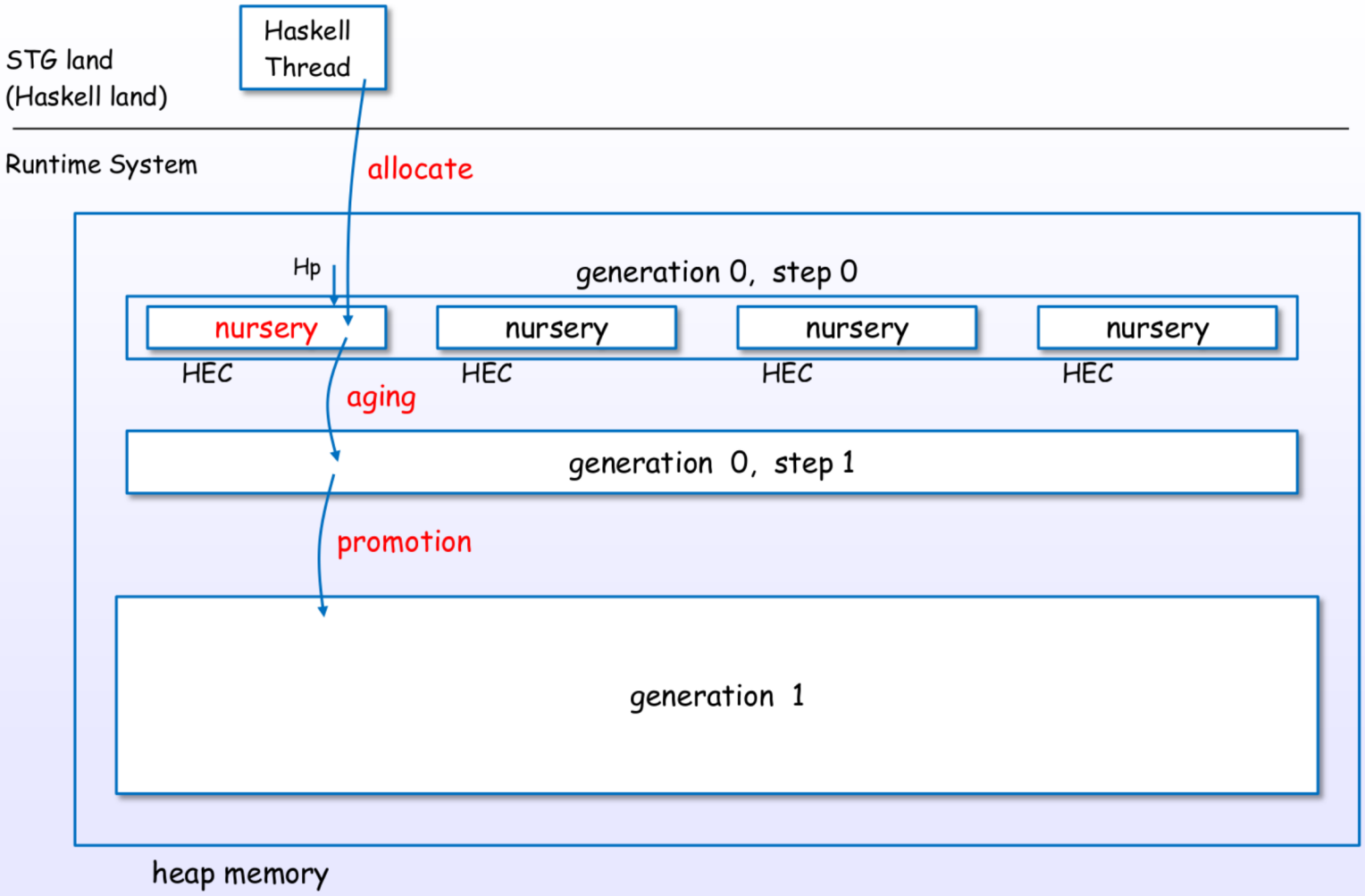


fast access using nursery for each processors

References : [5], [8], [9], [14], [C17], [C11], [19], [S17], [S16], [S23], [S22], [S14], [S17], [S16], [S25]

Threads and GC

GC, nursery, generation, aging, promotion

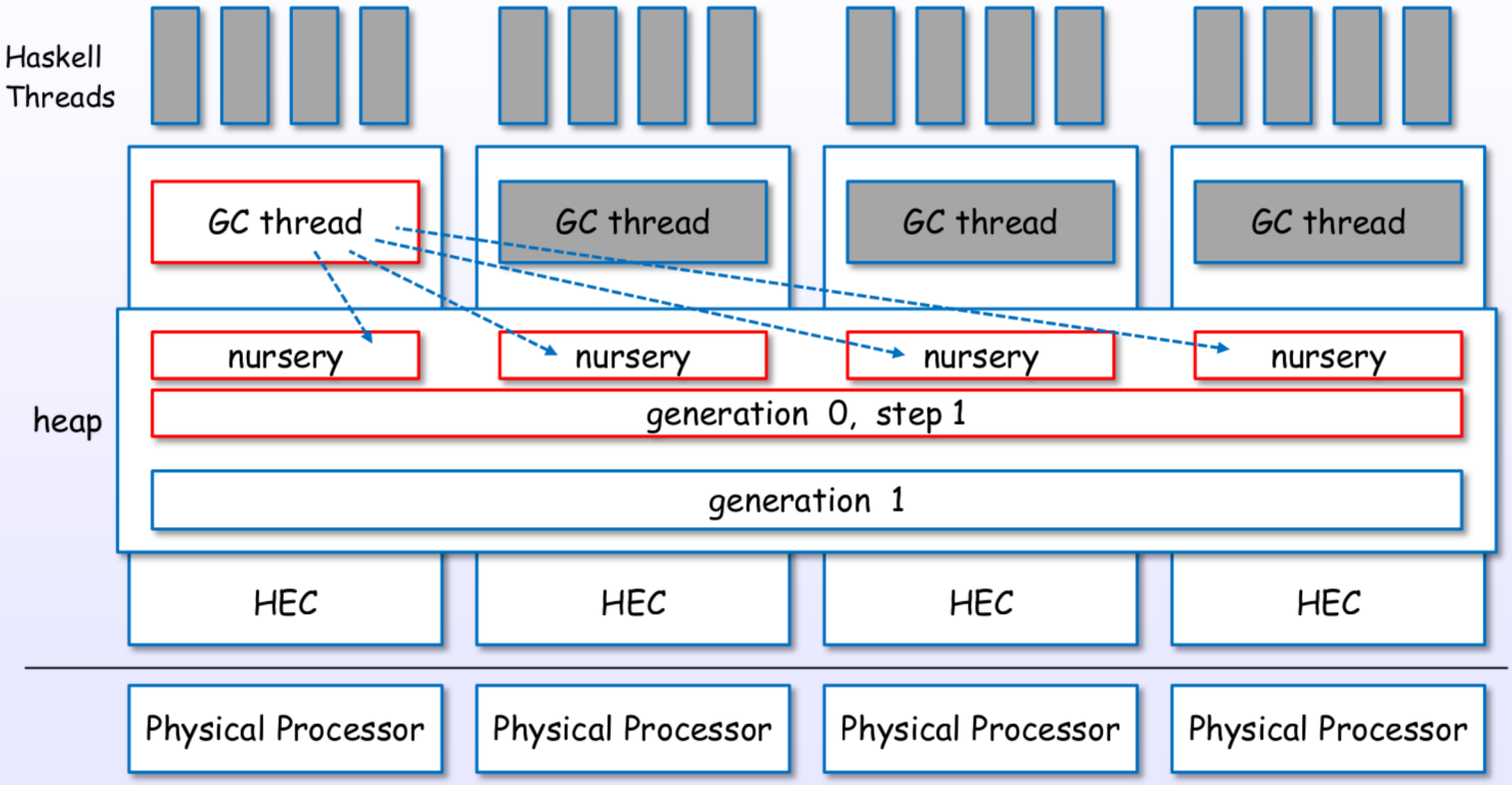


References : [8], [9], [15], [C13], [C11], [S25]

Threads and minor GC

sequential GC for young generation (minor GC)

"stop-the-world" GC

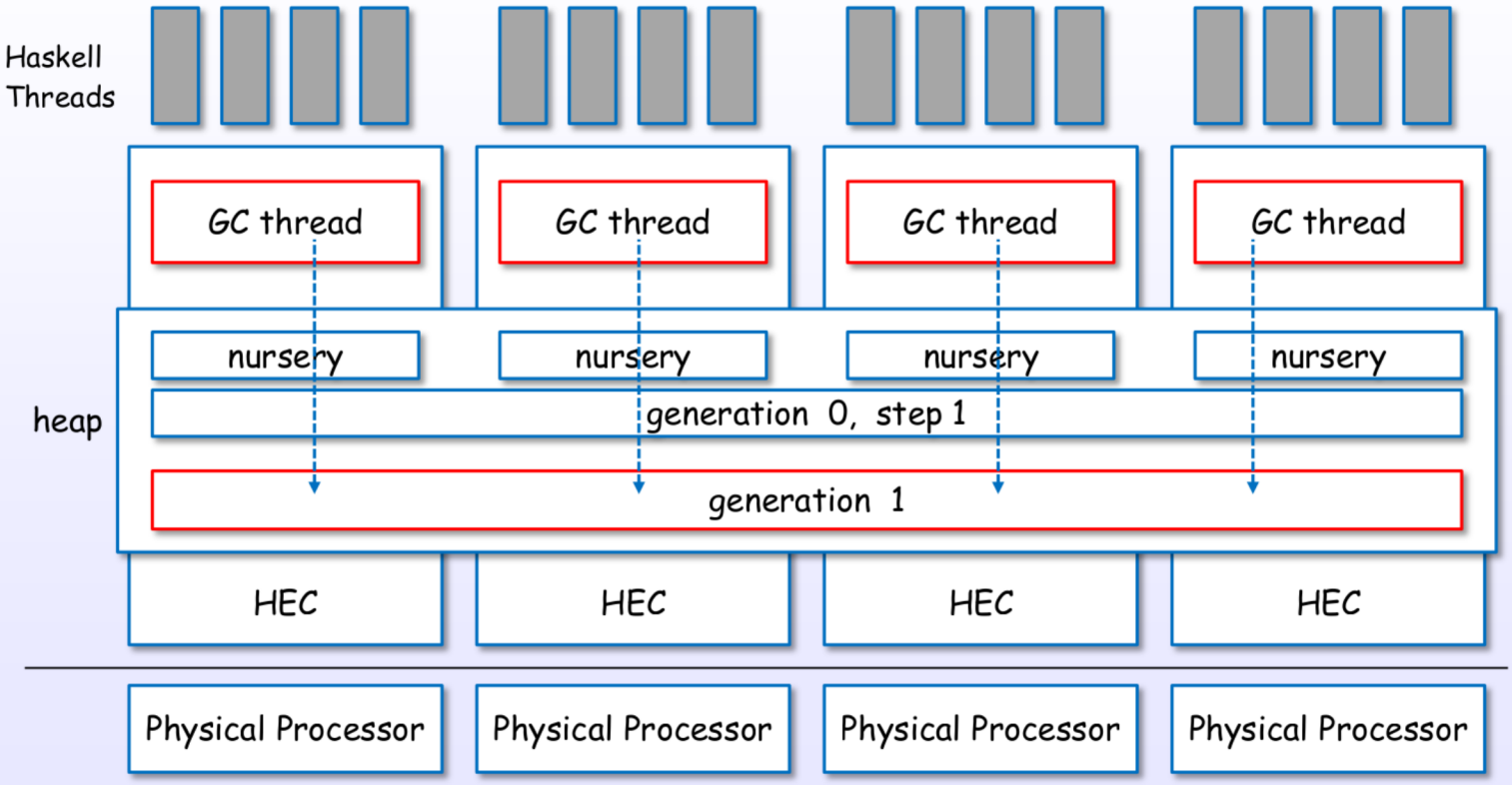


References : [8], [9], [15], [C13], [C11], [S25]

Threads and major GC

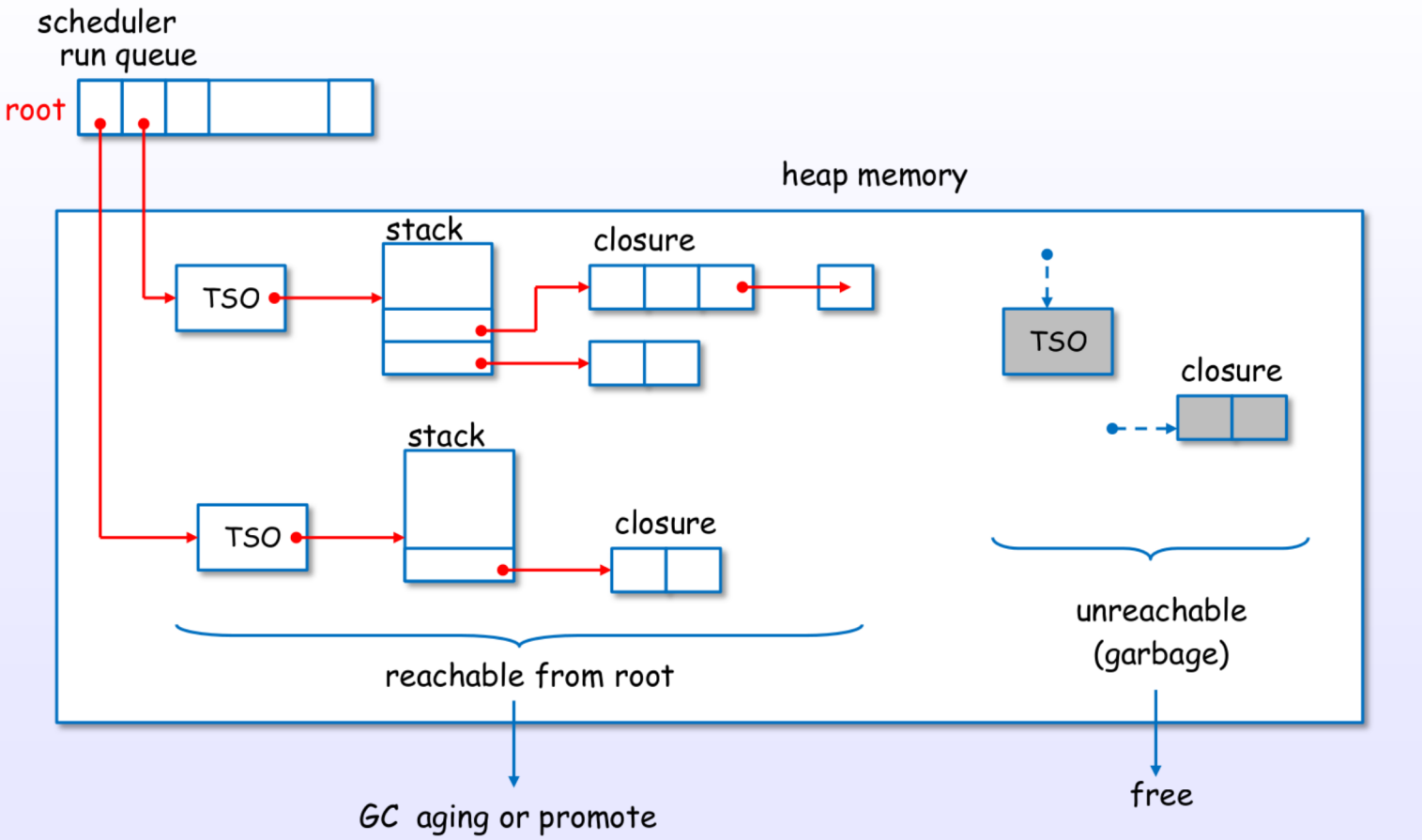
parallel GC for oldest generation (major GC)

"stop-the-world" GC



References : [8], [9], [15], [C13], [C11], [S25]

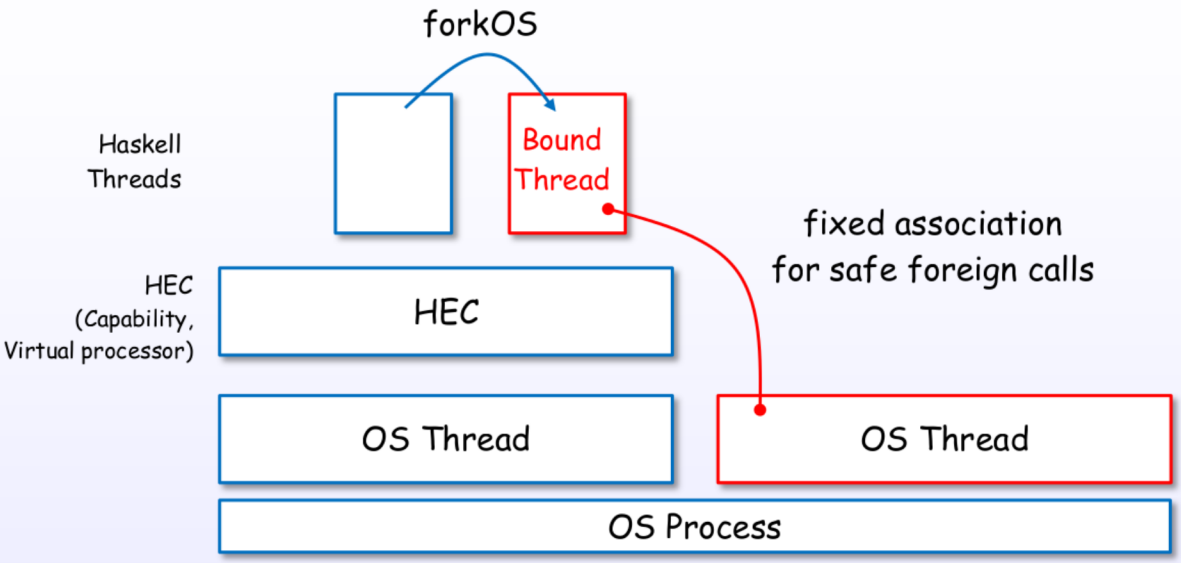
GC discover live objects from the root



References : [8], [9], [15], [C13], [C11], [S25]

Bound thread

A bound thread has a fixed associated OS Thread

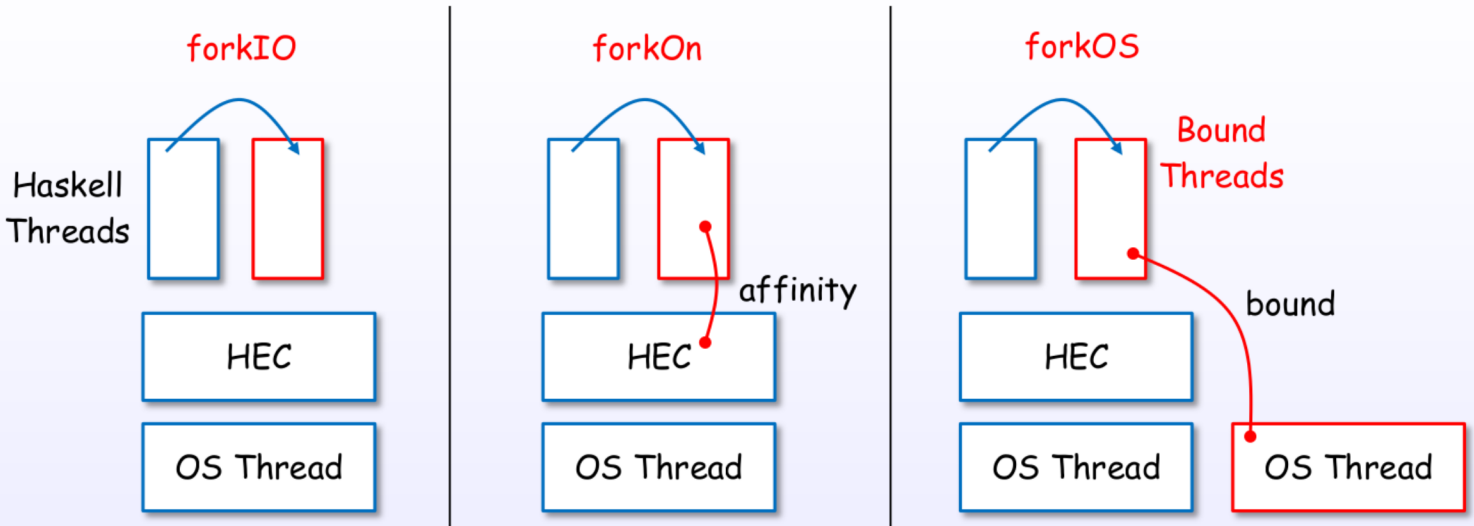


Foreign calls from a bound thread are all made by the same OS thread.
A bound thread is created using forkOS.

The main thread is bound thread.

References : [6], [5], [8], [9], [14], [C17], [19], [S17], [S16], [S23], [S22]

forkIO, forkOn, forkOS



create a haskell unbound thread

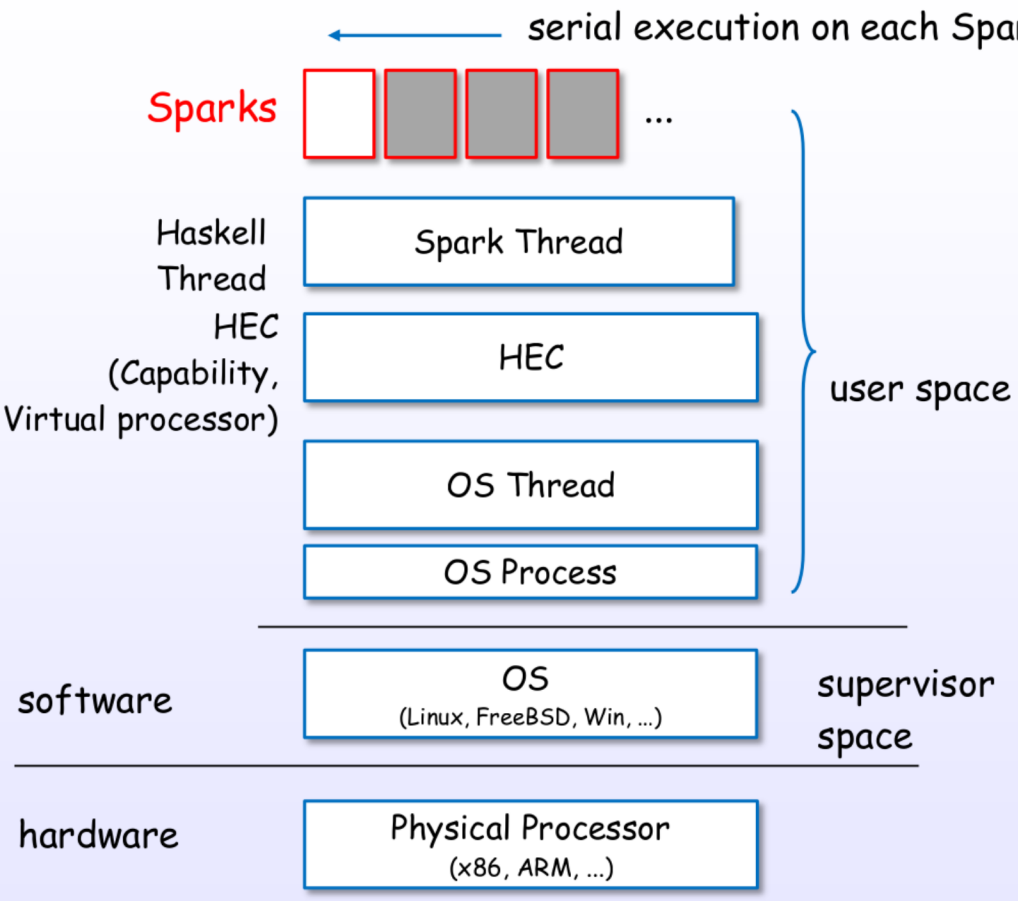
create a haskell unbound thread
on the specified HEC

create a haskell bound thread
and an OS thread

References : [6], [5], [8], [9], [14], [C17], [19], [S17], [S16], [S23], [S22]

Spark

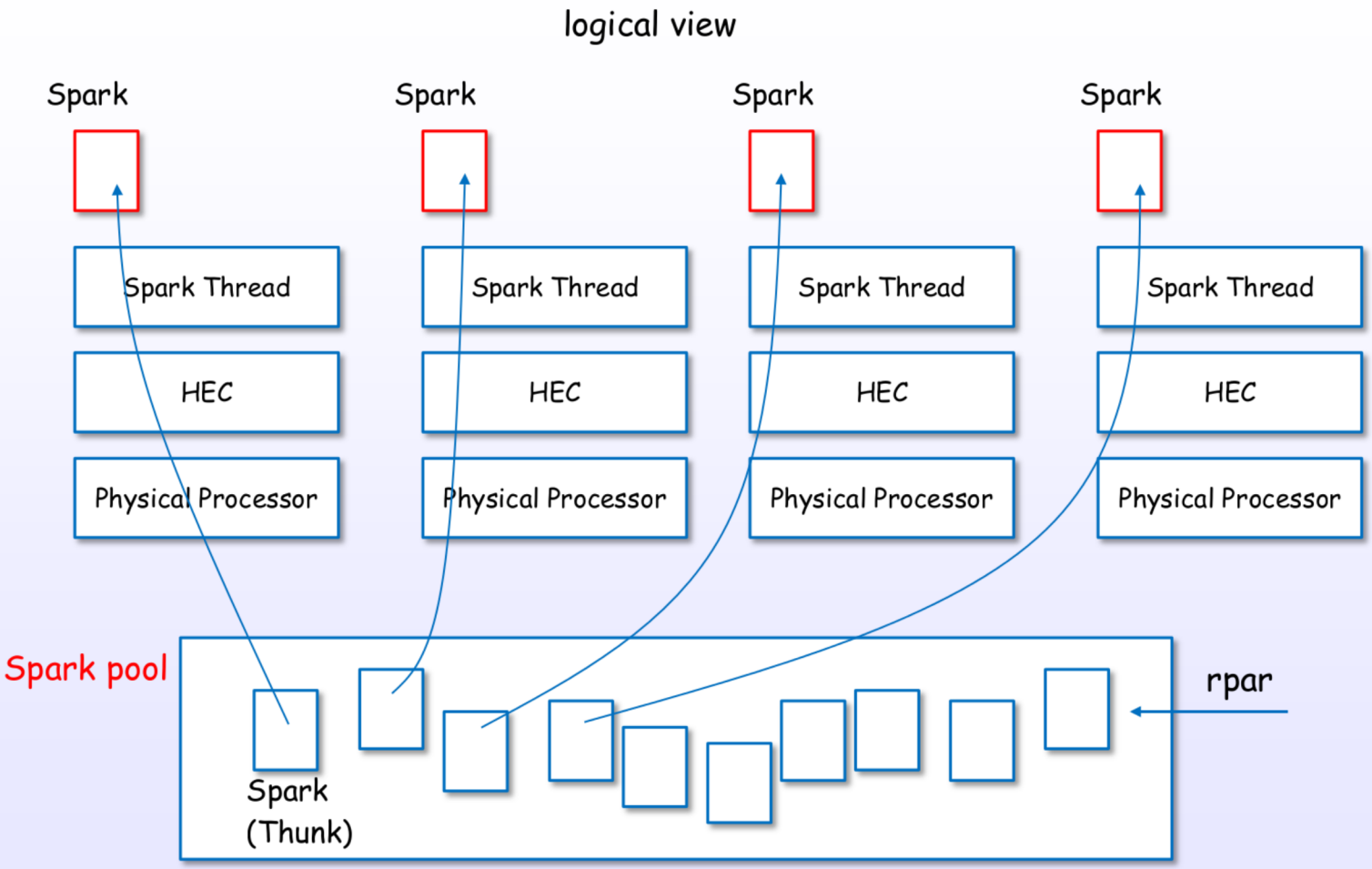
Spark layer



Spark Threads are generated on idle HECs.

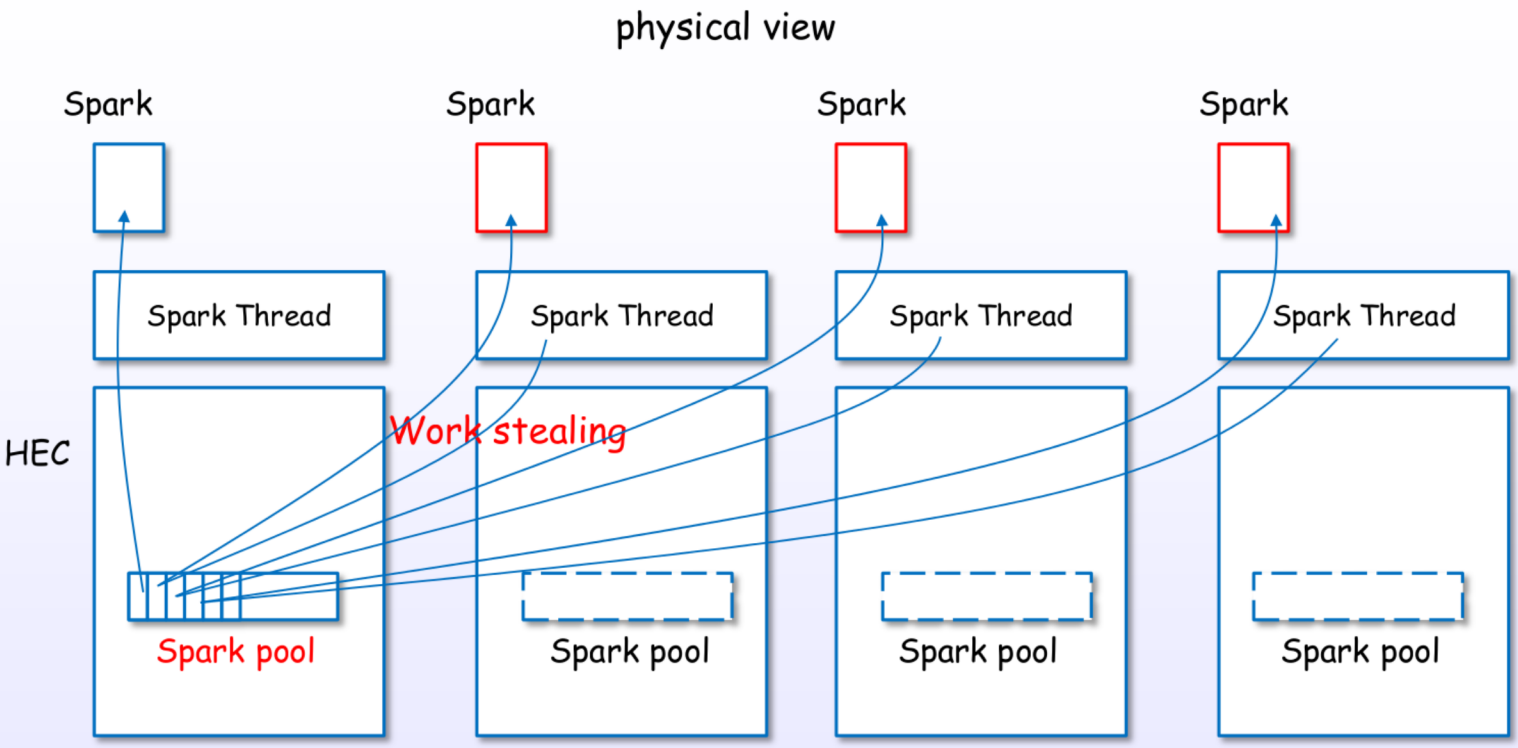
References : [C17], [19], [S17], [S26], [S27], [S33], [S12]

Sparks and Spark pool



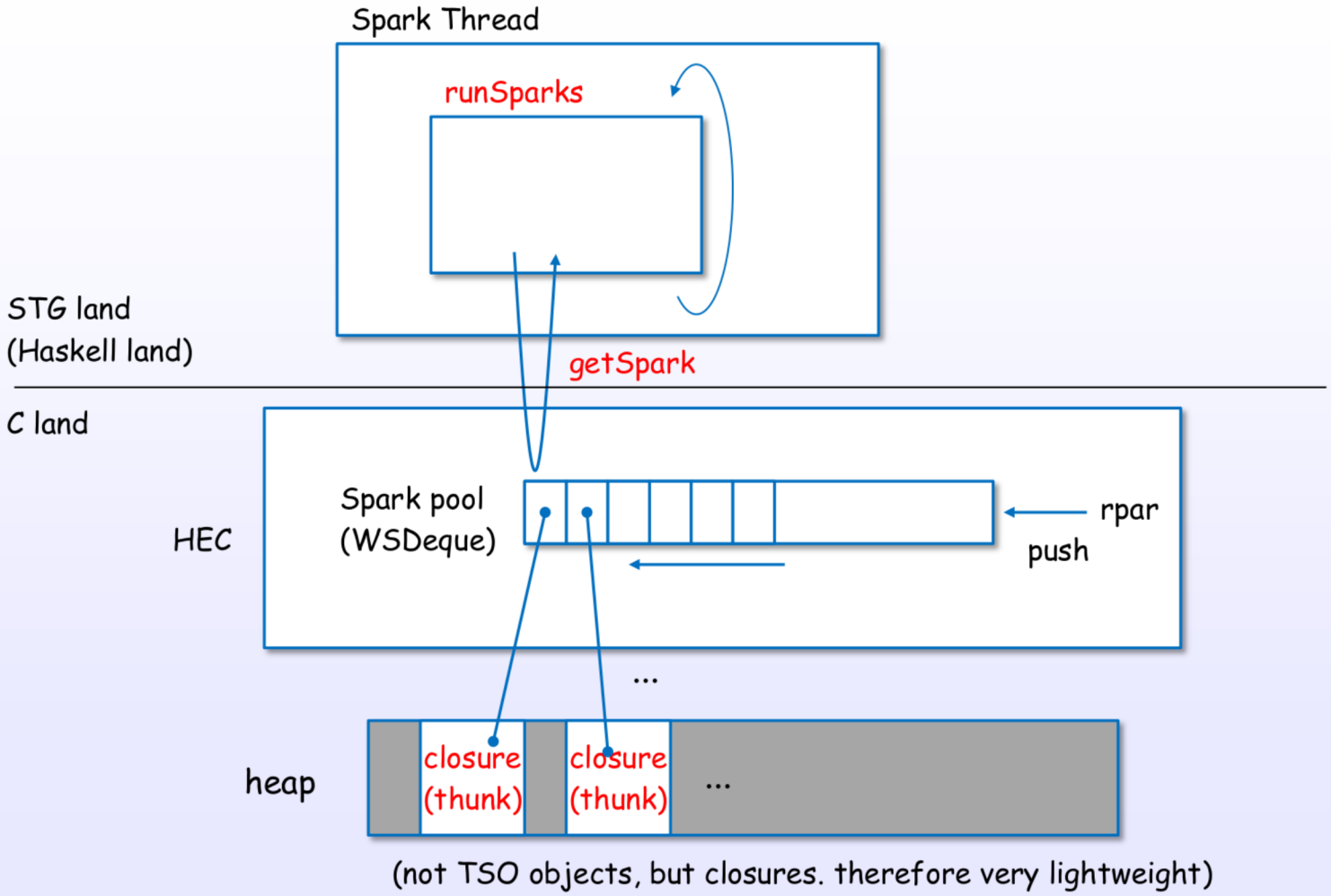
References : [C17], [19], [S17], [S26], [S27], [S33], [S12]

Spark pool and work stealing



References : [C17], [19], [S17], [S26], [S27], [S33], [S12]

Sparks and closures

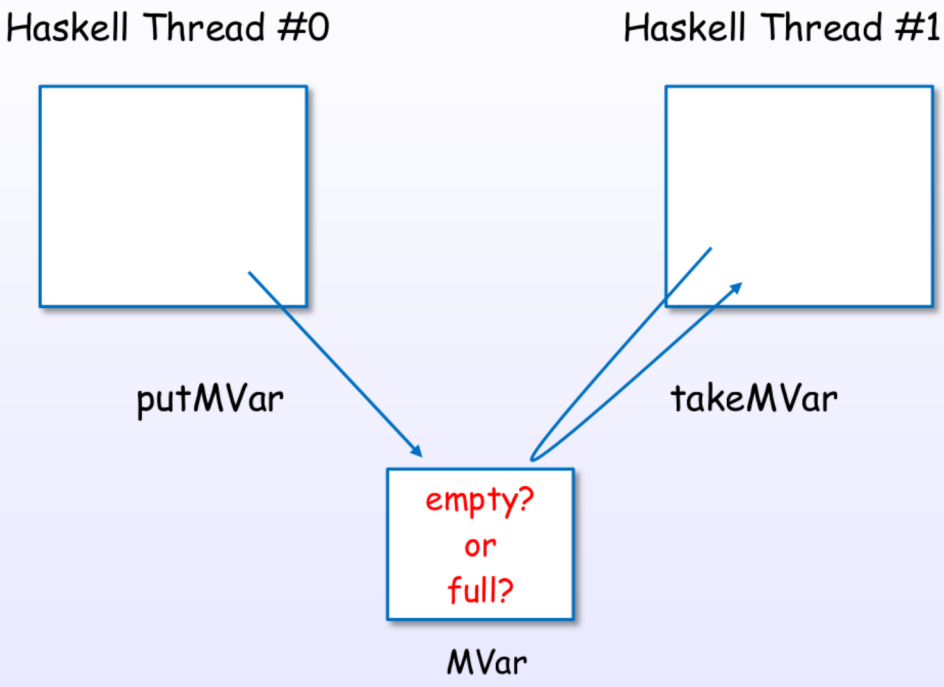


(not TSO objects, but closures. therefore very lightweight)

References : [C17], [19], [S17], [S26], [S27], [S33], [S12]

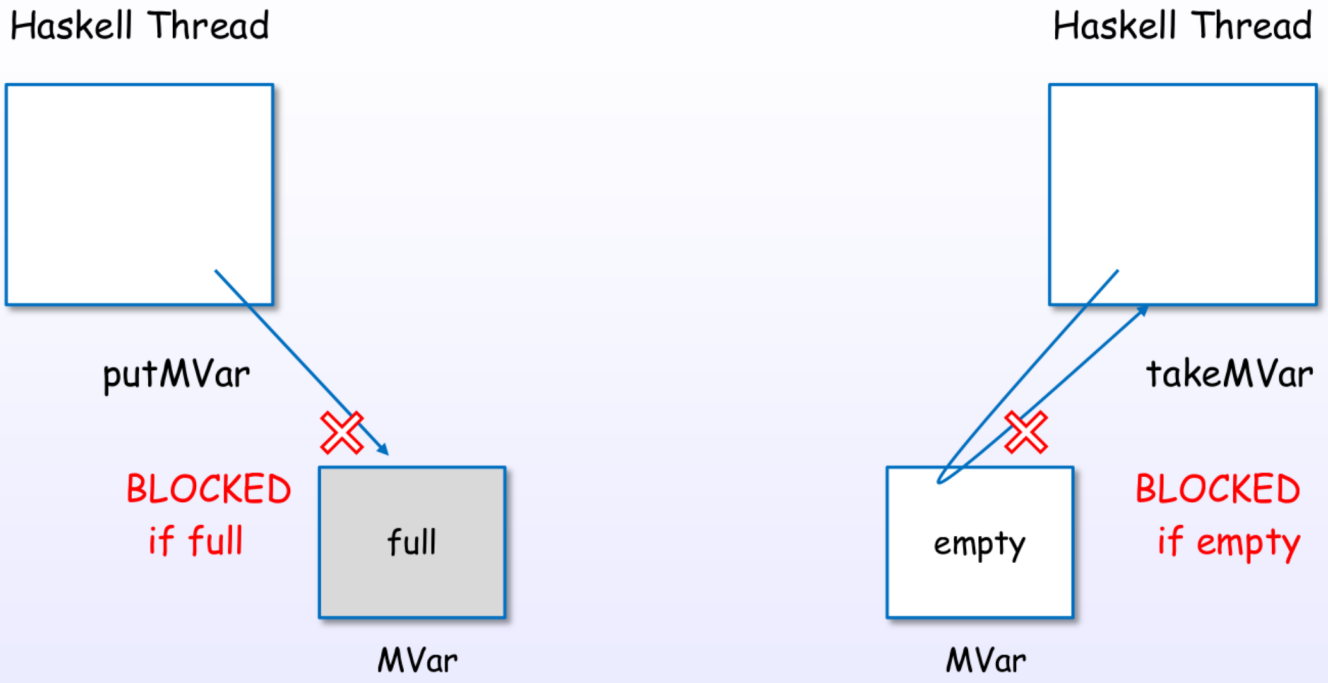
MVar

MVar



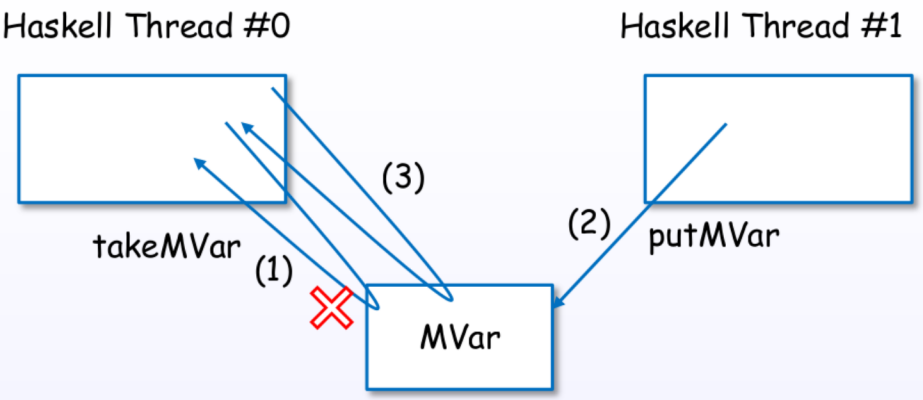
References : [16], [18], [19], [S31], [S12]

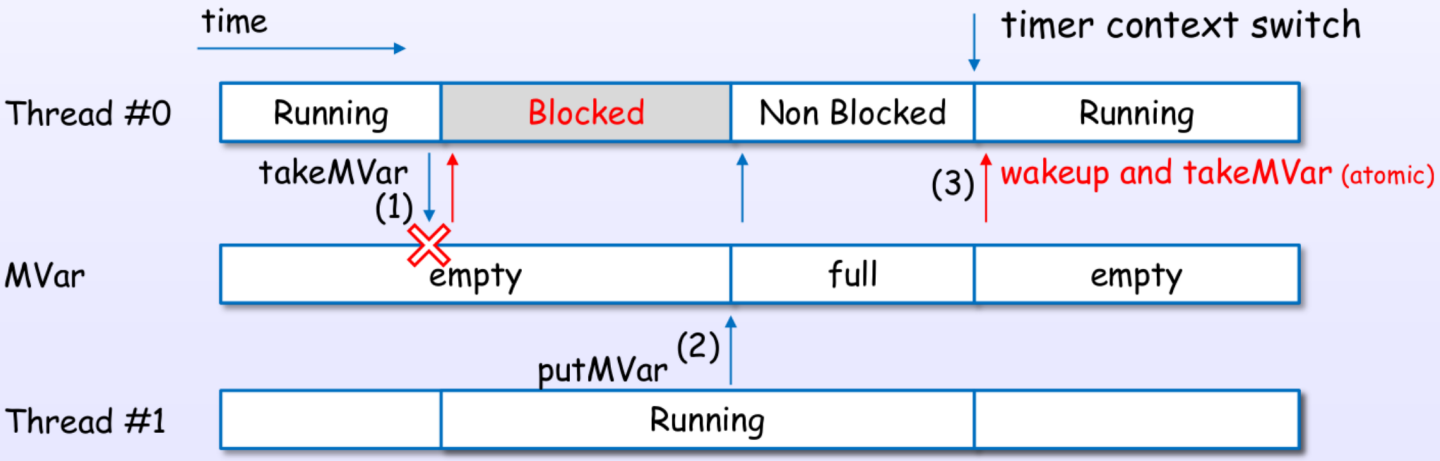
MVar and blocking



References : [16], [18], [19], [S31], [S12]

MVar example





* single core case

References : [16], [18], [19], [S31], [S12]

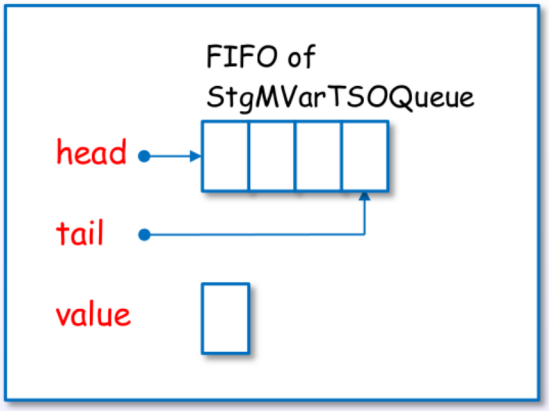
MVar object view

User view

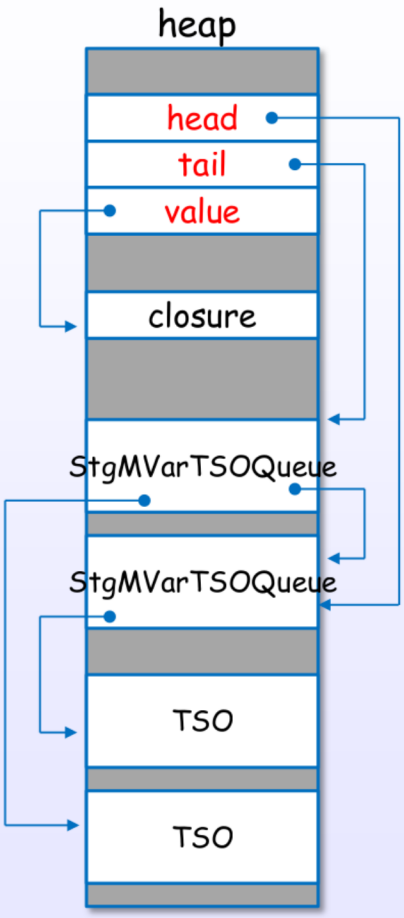
MVar

empty?
or
full?

logical MVar object

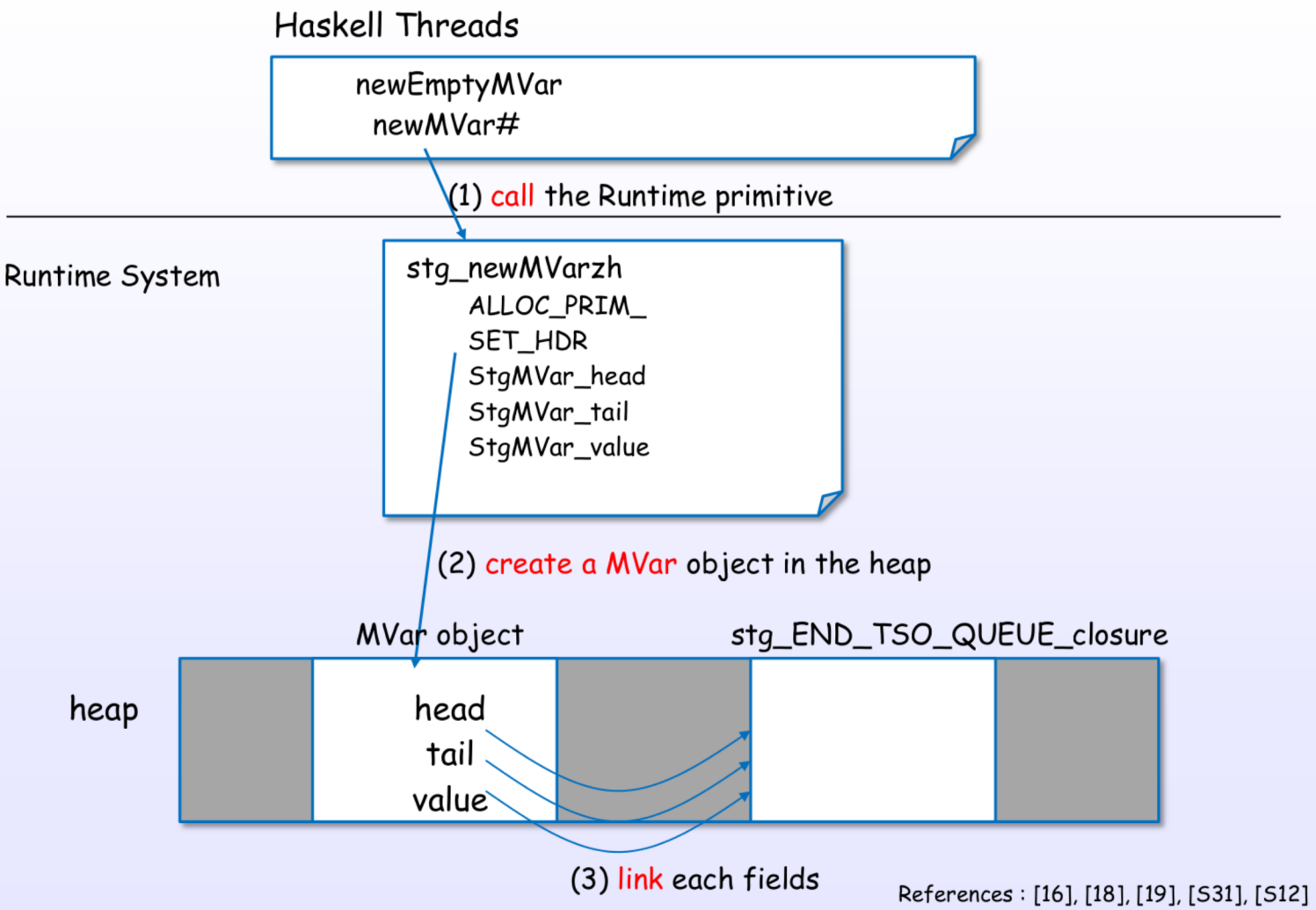


physical MVar object



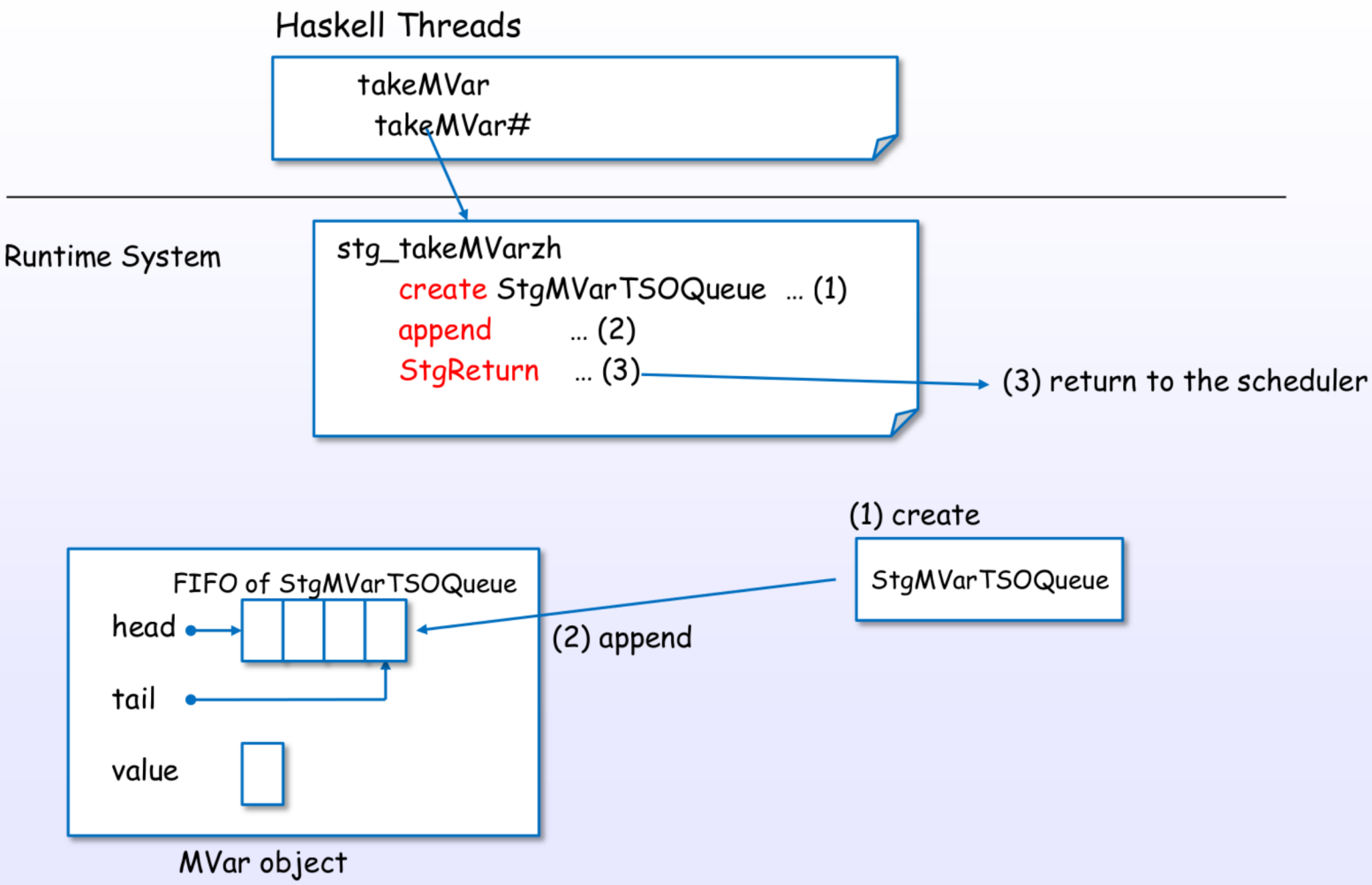
References : [16], [18], [19], [S31], [S12]

newEmptyMVar



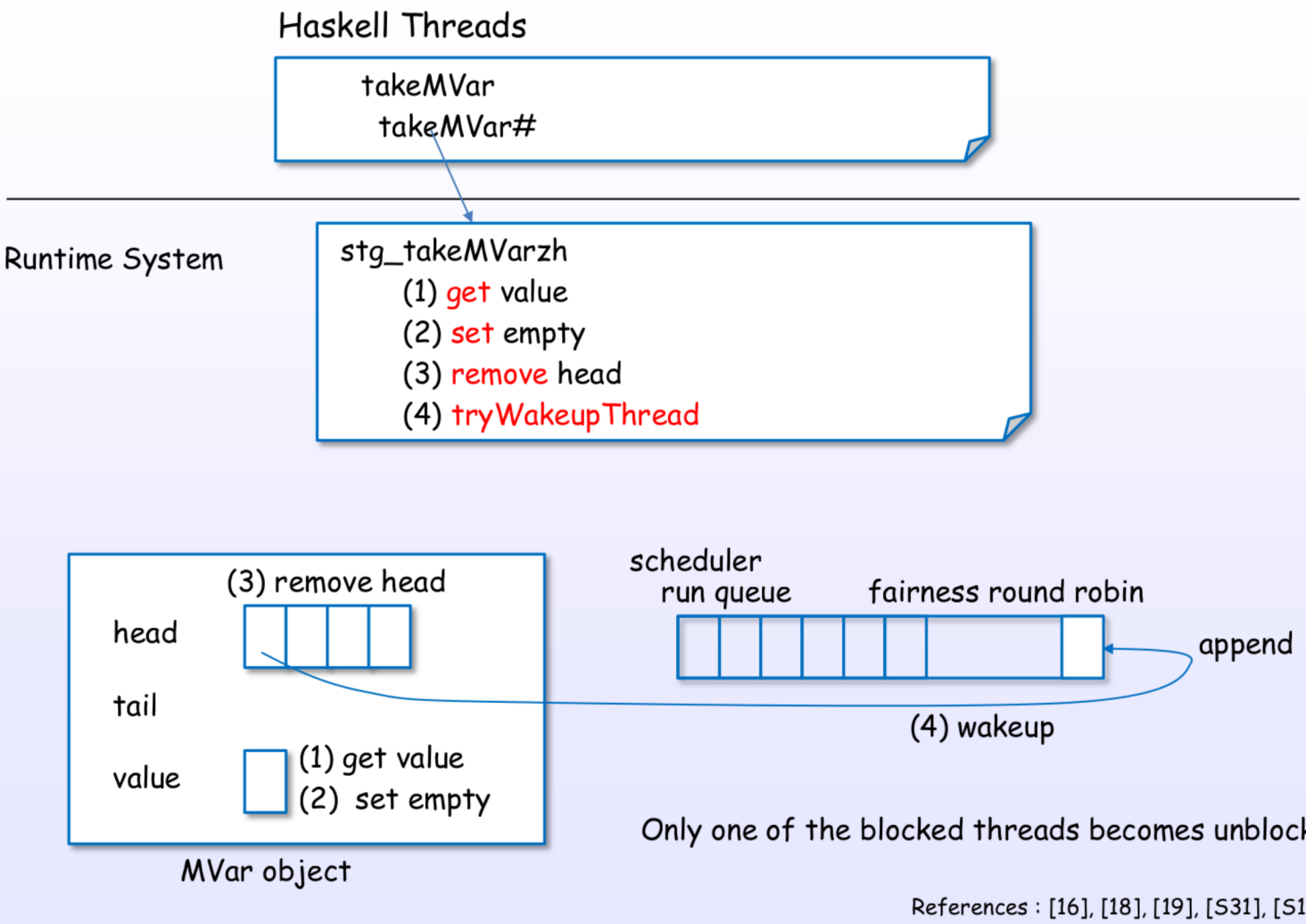
References : [16], [18], [19], [S31], [S12]

takeMVar (empty case)



References : [16], [18], [19], [S31], [S12]

takeMVar (full case)



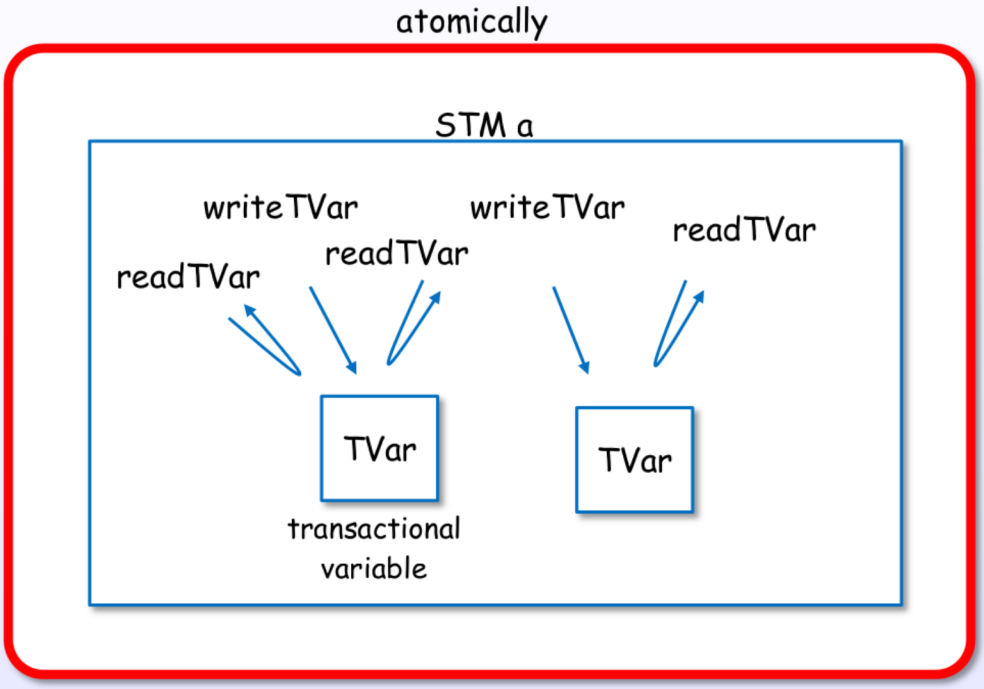
Only one of the blocked threads becomes unblocked.

References : [16], [18], [19], [S31], [S12]

Software transactional memory

Create a atomic block using atomically

atomically :: STM a -> IO a

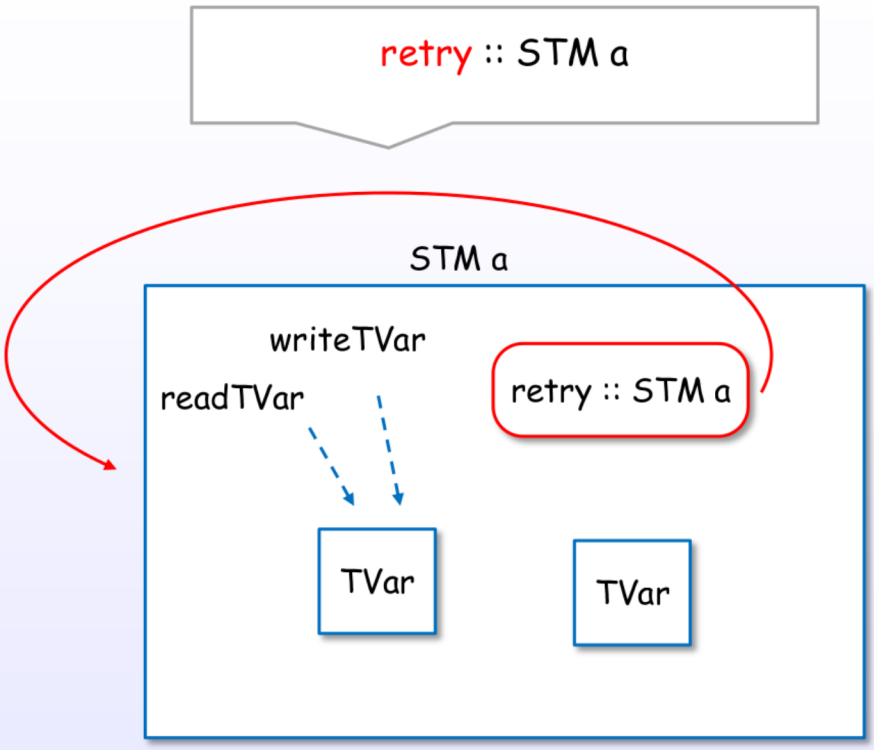


Create and evaluate a composable "atomic block"

Atomic block = All or Nothing

References : [17], [19], [20], [C18], [S12], [S28]

Rollback and blocking control using retry

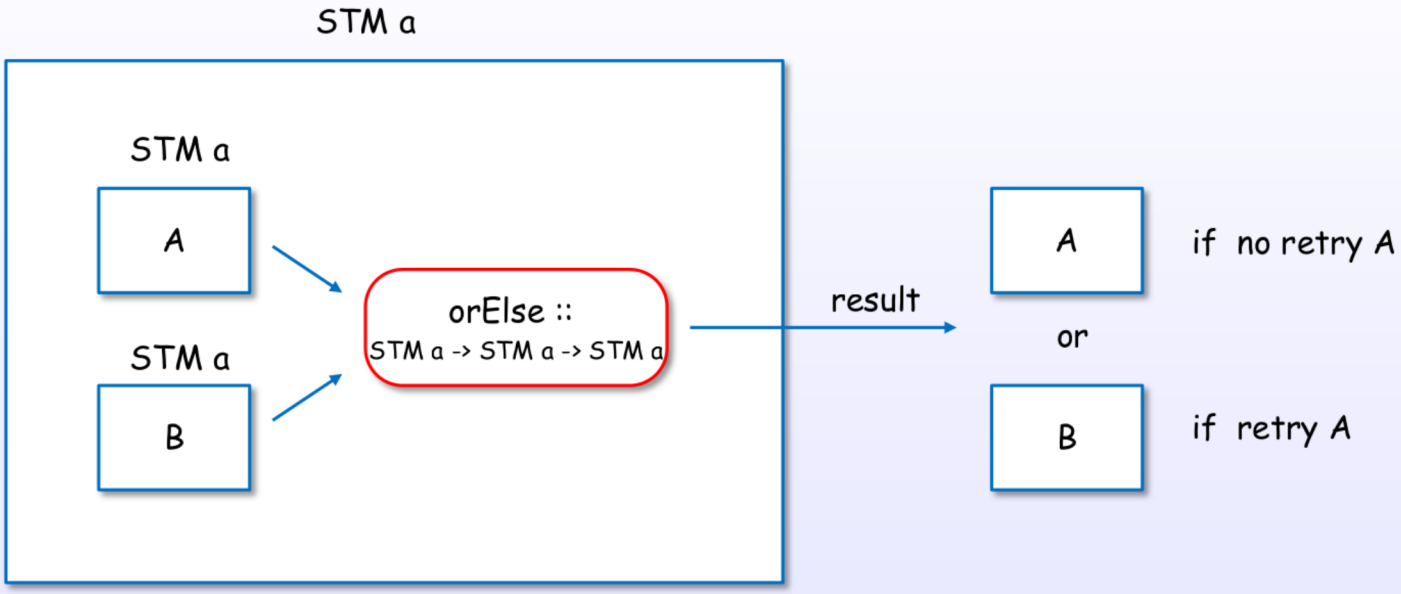


Discard, blocking and try again

References : [17], [19], [20], [C18], [S12], [S28]

Compose OR case using orElse

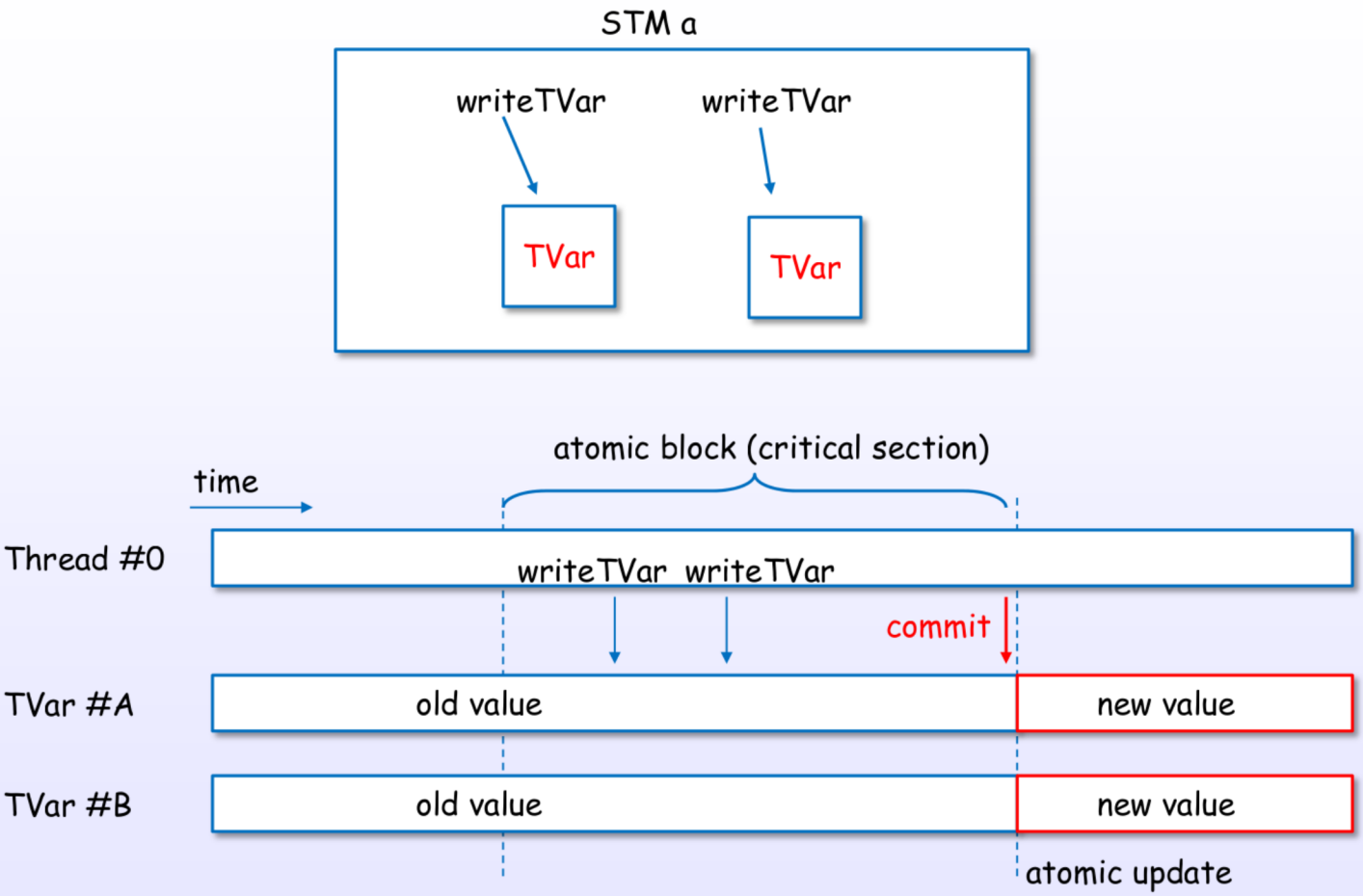
```
orElse :: STM a -> STM a -> STM a
```



A or B or Nothing

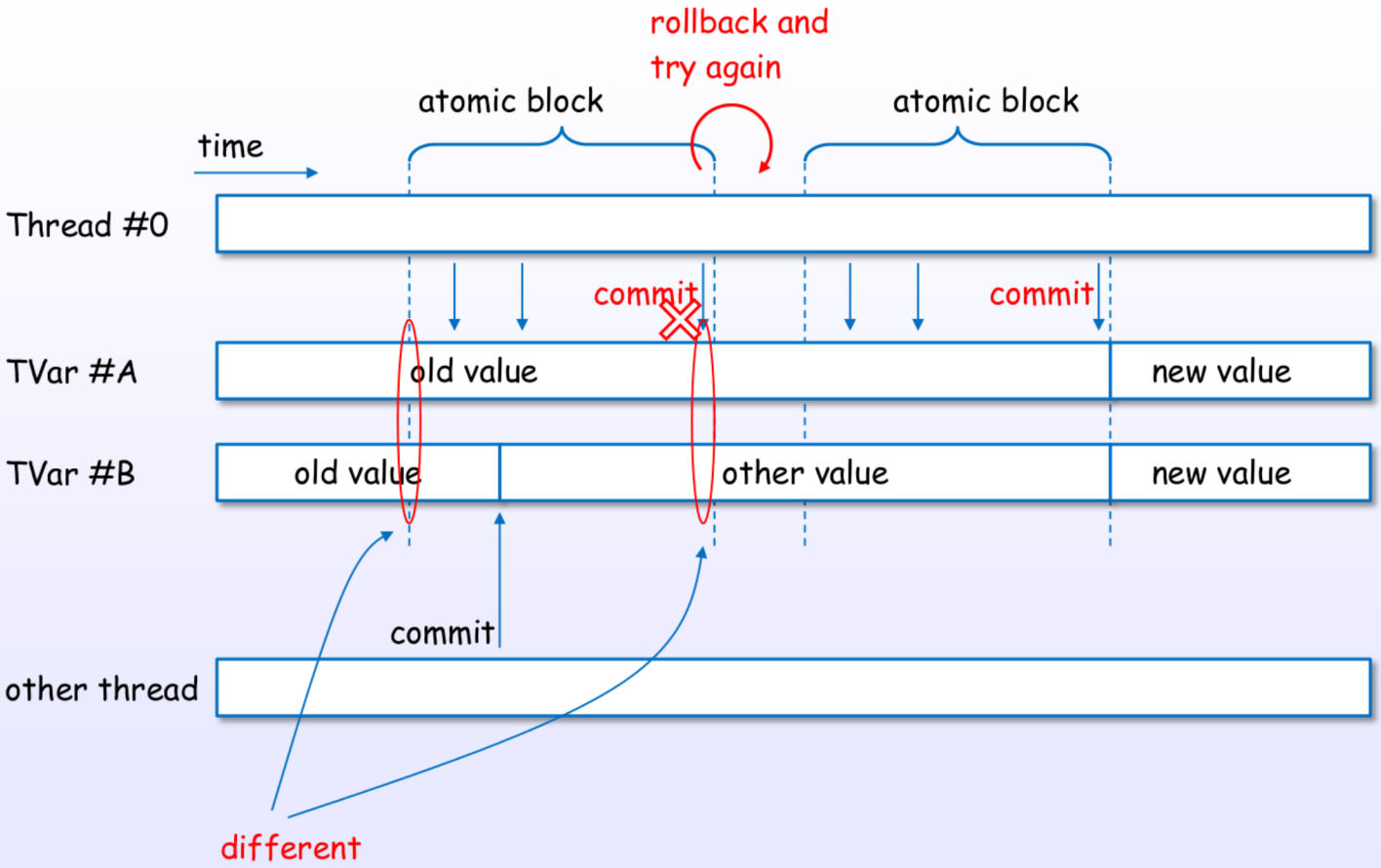
References : [17], [19], [20], [C18], [S12], [S28]

STM, TVar example (normal case)



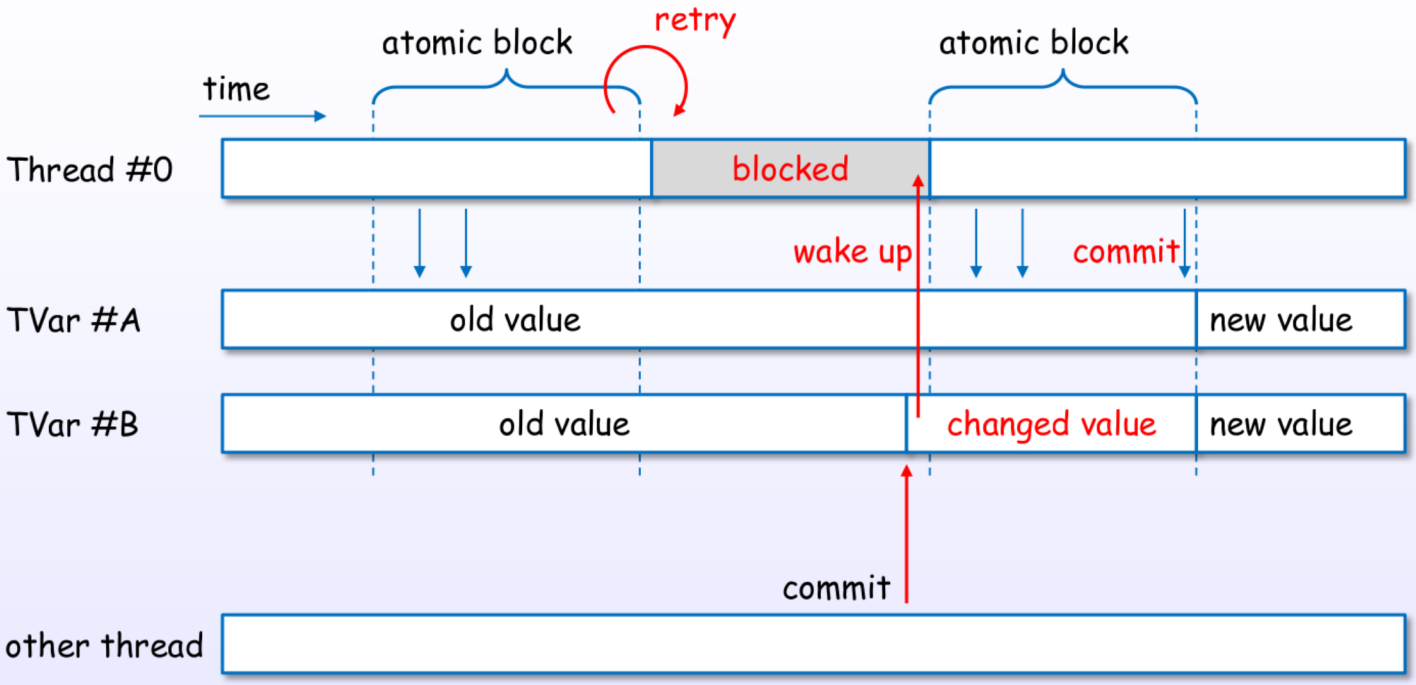
References : [17], [19], [20], [C18], [S12], [S28]

STM, TVar example (conflict case)



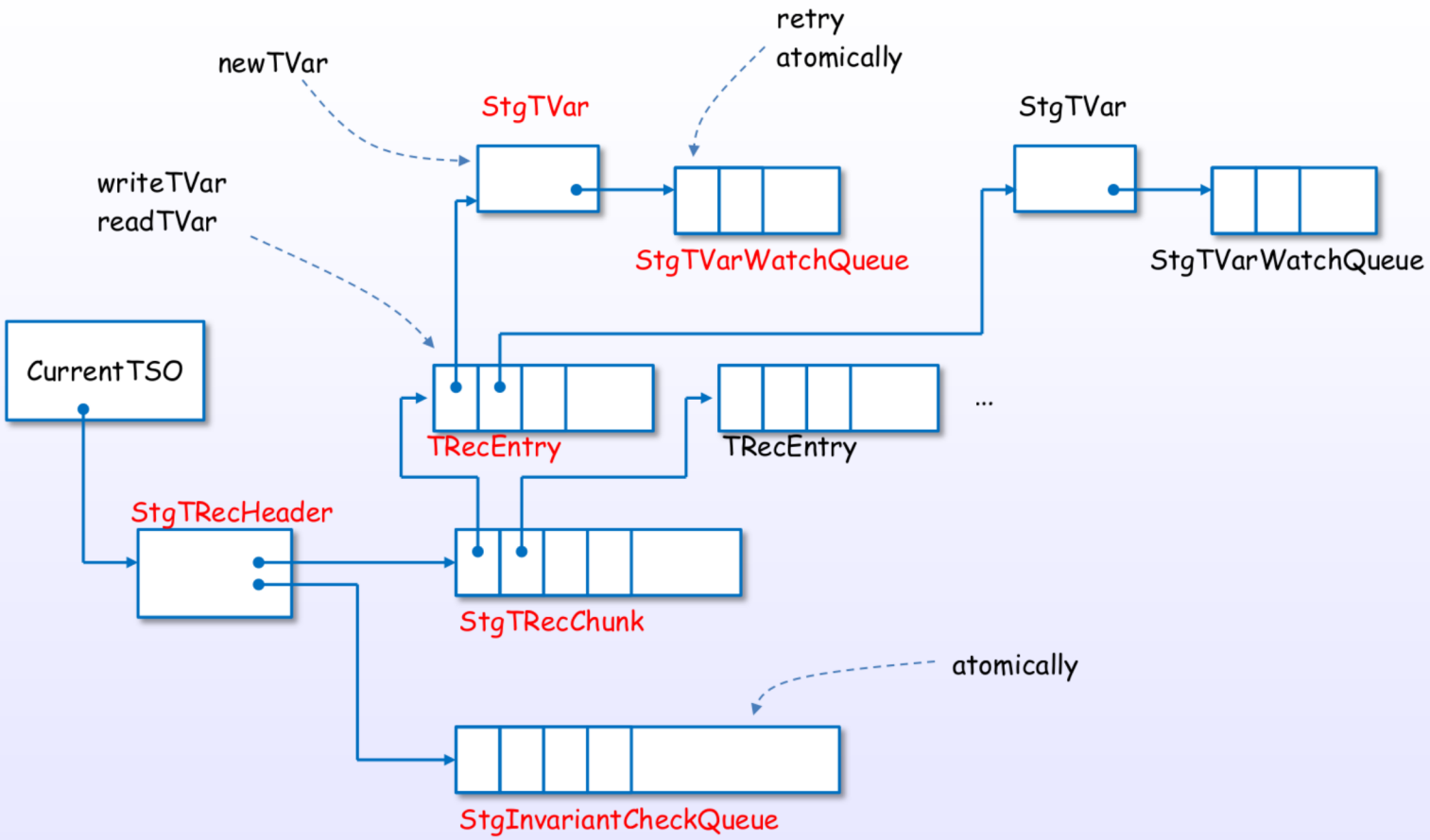
References : [17], [19], [20], [C18], [S12], [S28]

retry example



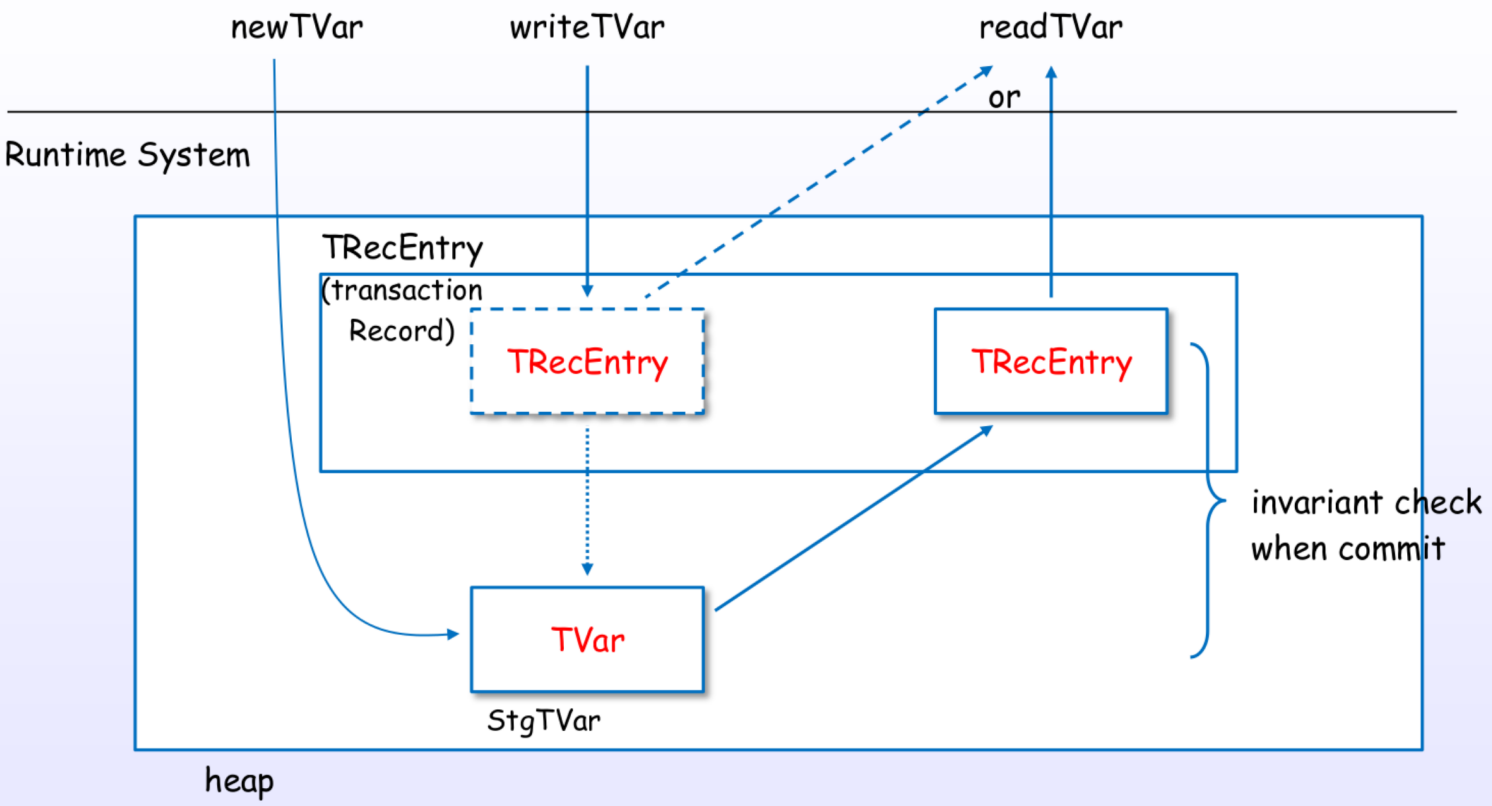
References : [17], [19], [20], [C18], [S12], [S28]

STM, TVar data structure



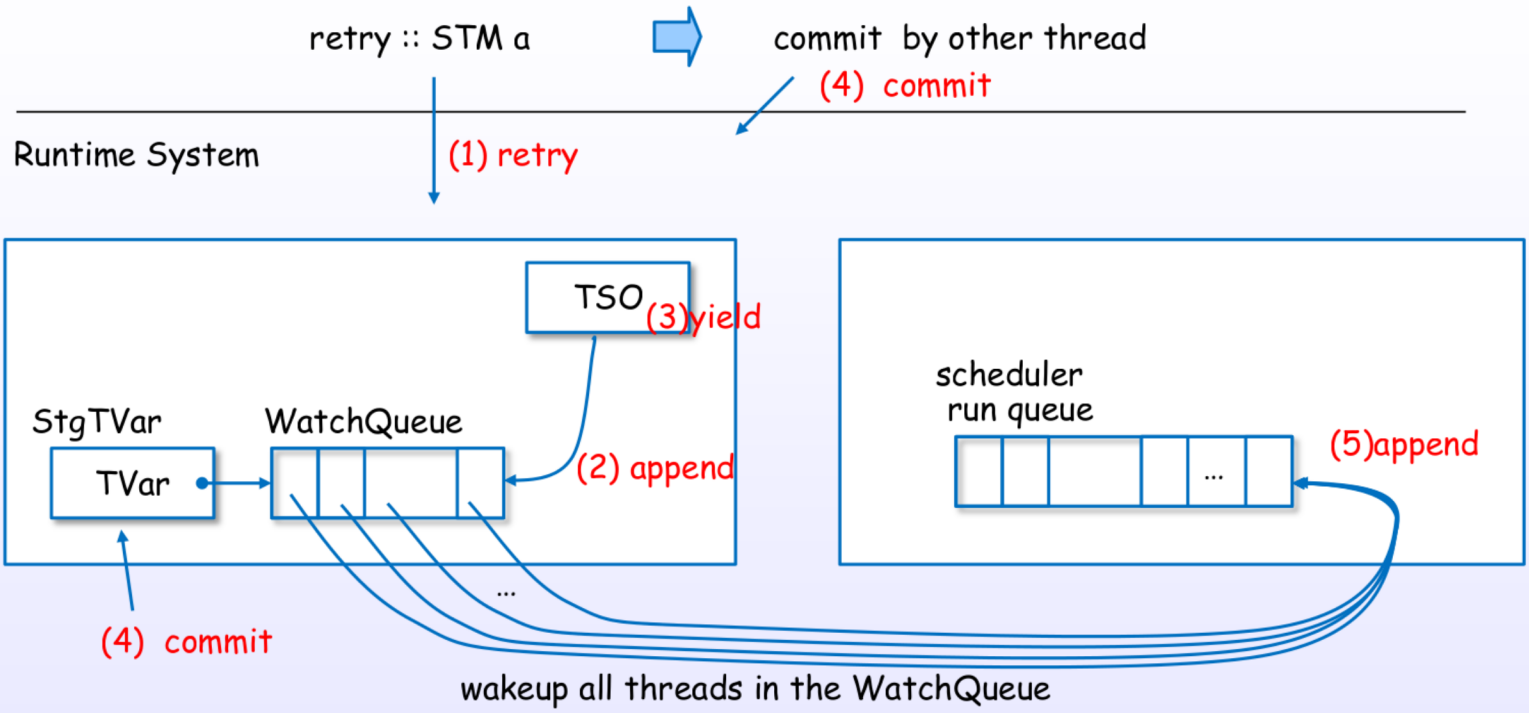
References : [17], [19], [20], [C18], [S12], [S28]

newTVar, writeTVar, readTVar



References : [17], [19], [20], [C18], [S12], [S28]

block by retry, wake up by commit

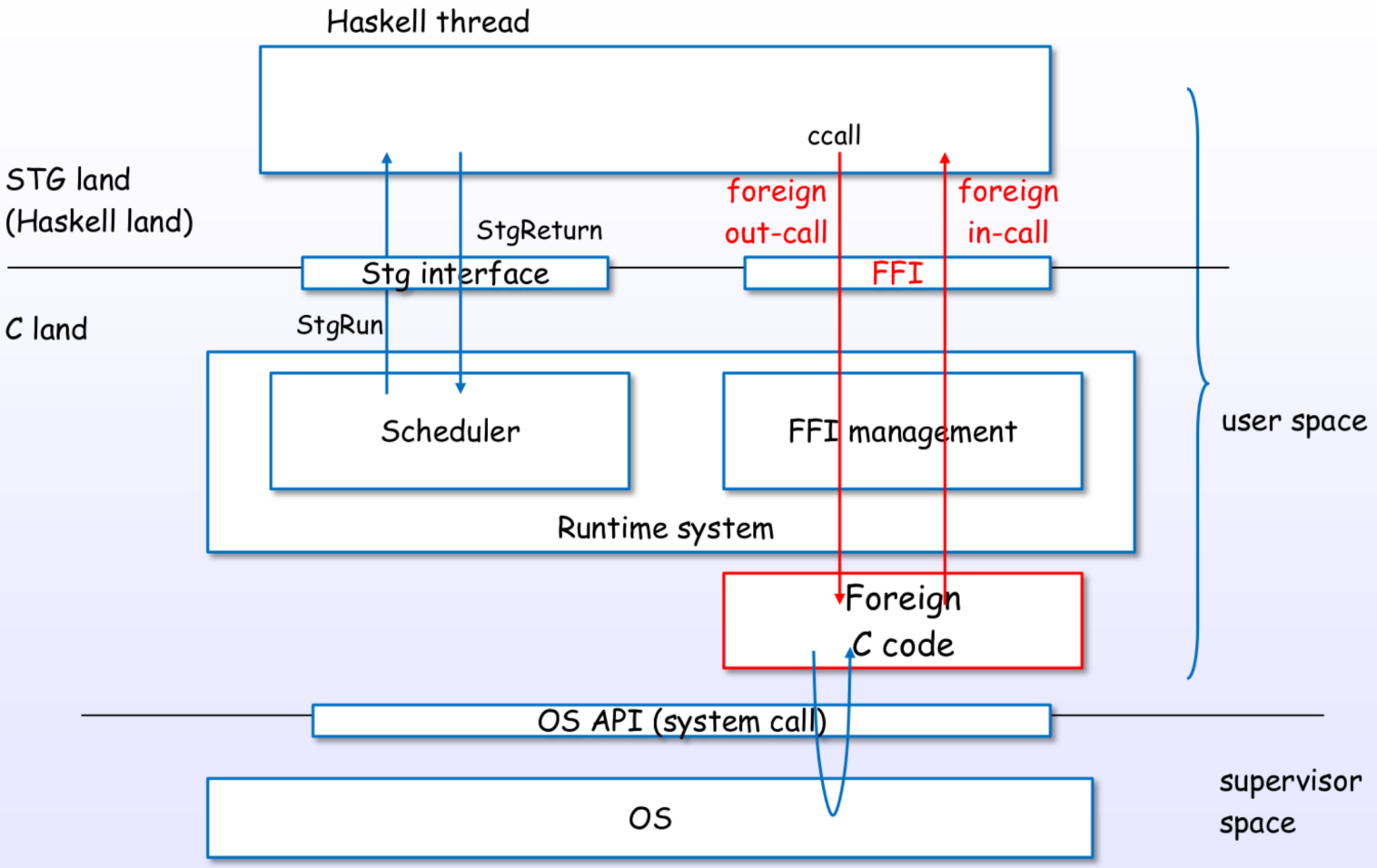


no guarantee of fairness,
because the RTS has to run all the blocked transaction.

References : [17], [19], [20], [C18], [S12], [S28]

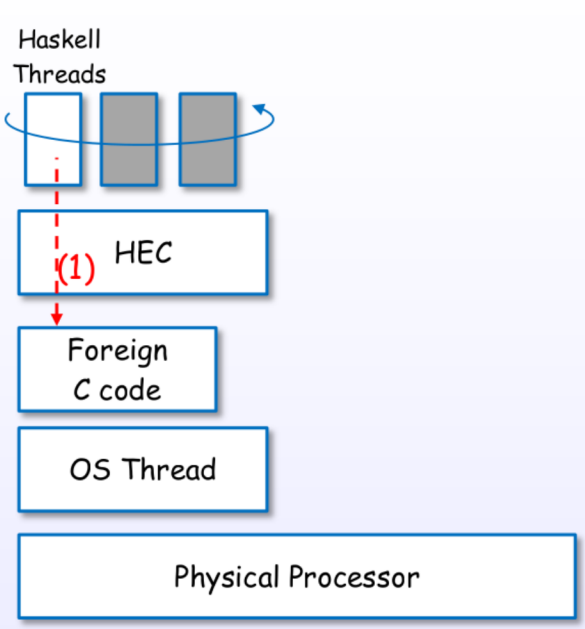
FFI

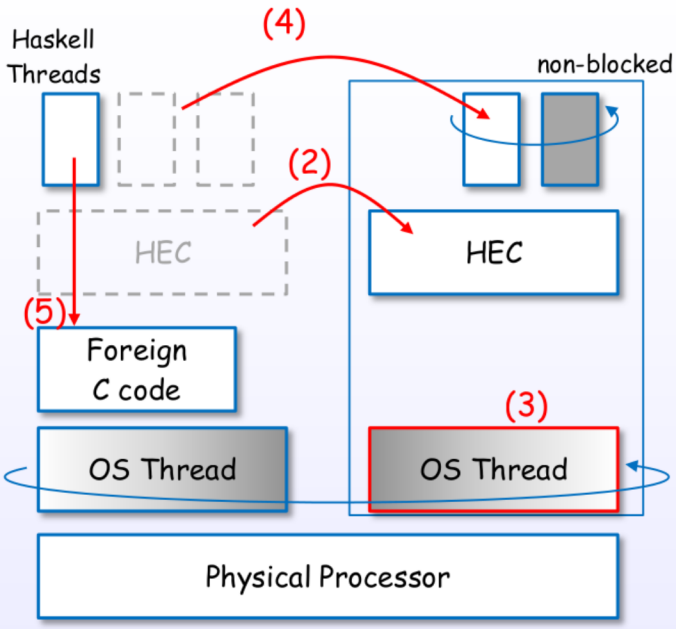
FFI (Foreign Function Interface)



References : [6], [11], [20], [S39], [S38], [S37], [S36], [S40]

FFI and OS Threads





(1) a safe foreign call (FFI)

(2) move the HEC to other OS thread

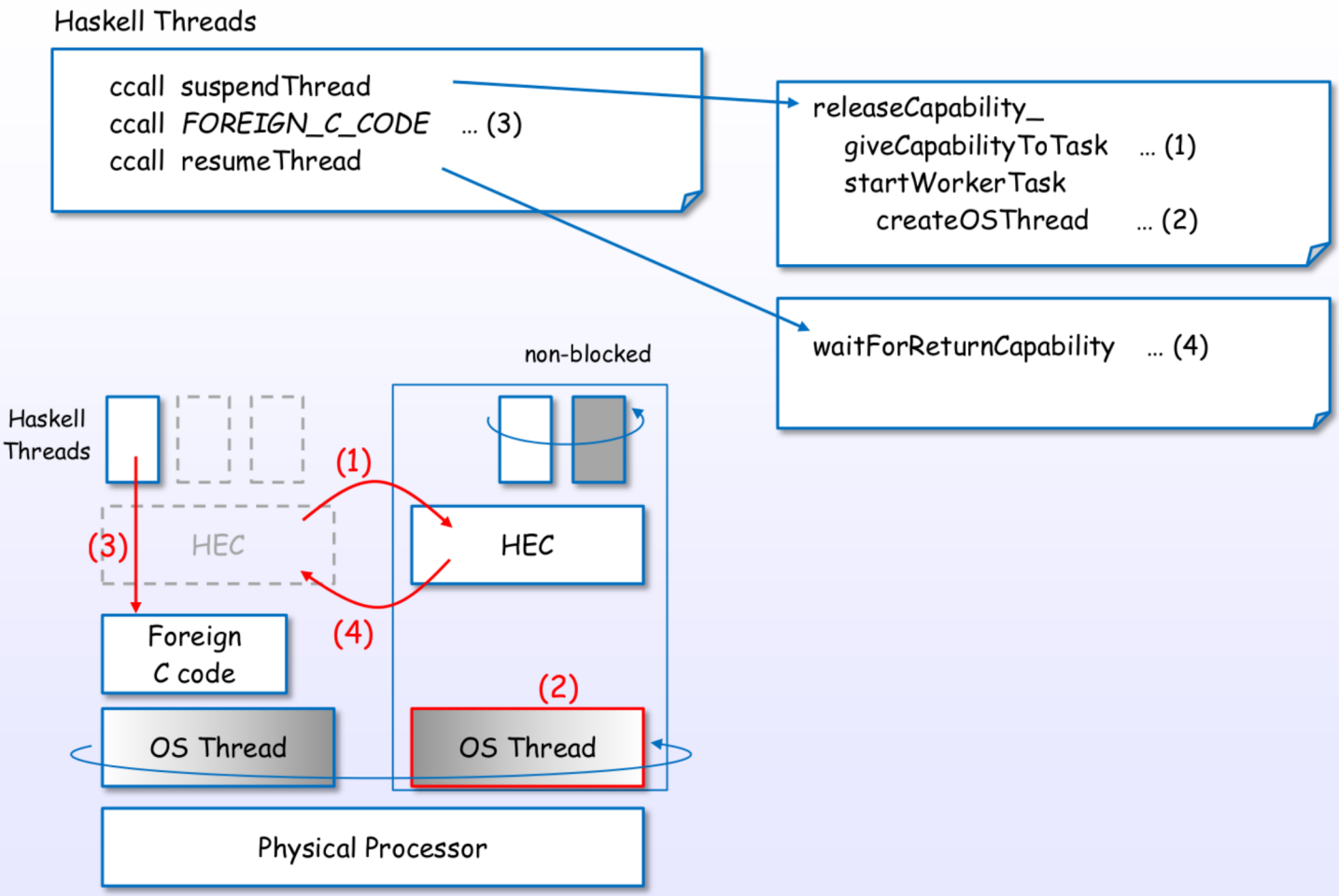
(3) spawn or draw an OS thread

(4) move Haskell threads

(5) call the foreign C code

References : [6], [11], [20], [S39], [S38], [S37], [S36], [S40]

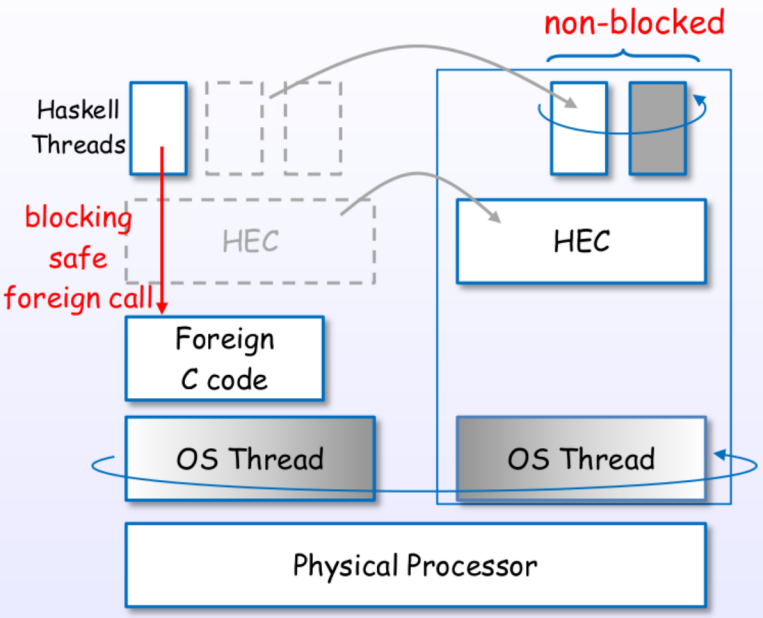
A safe foreign call (code)



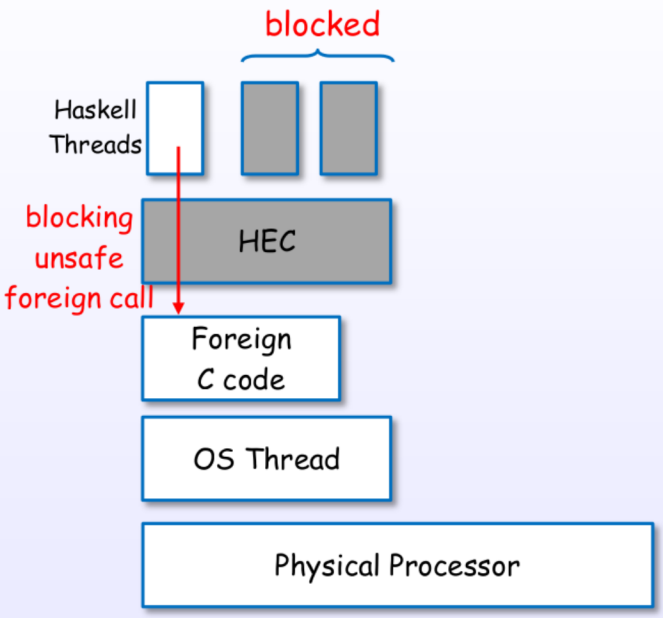
References : [6], [11], [20], [S39], [S38], [S37], [S36], [S40]

a safe and an unsafe foreign call

a safe foreign call



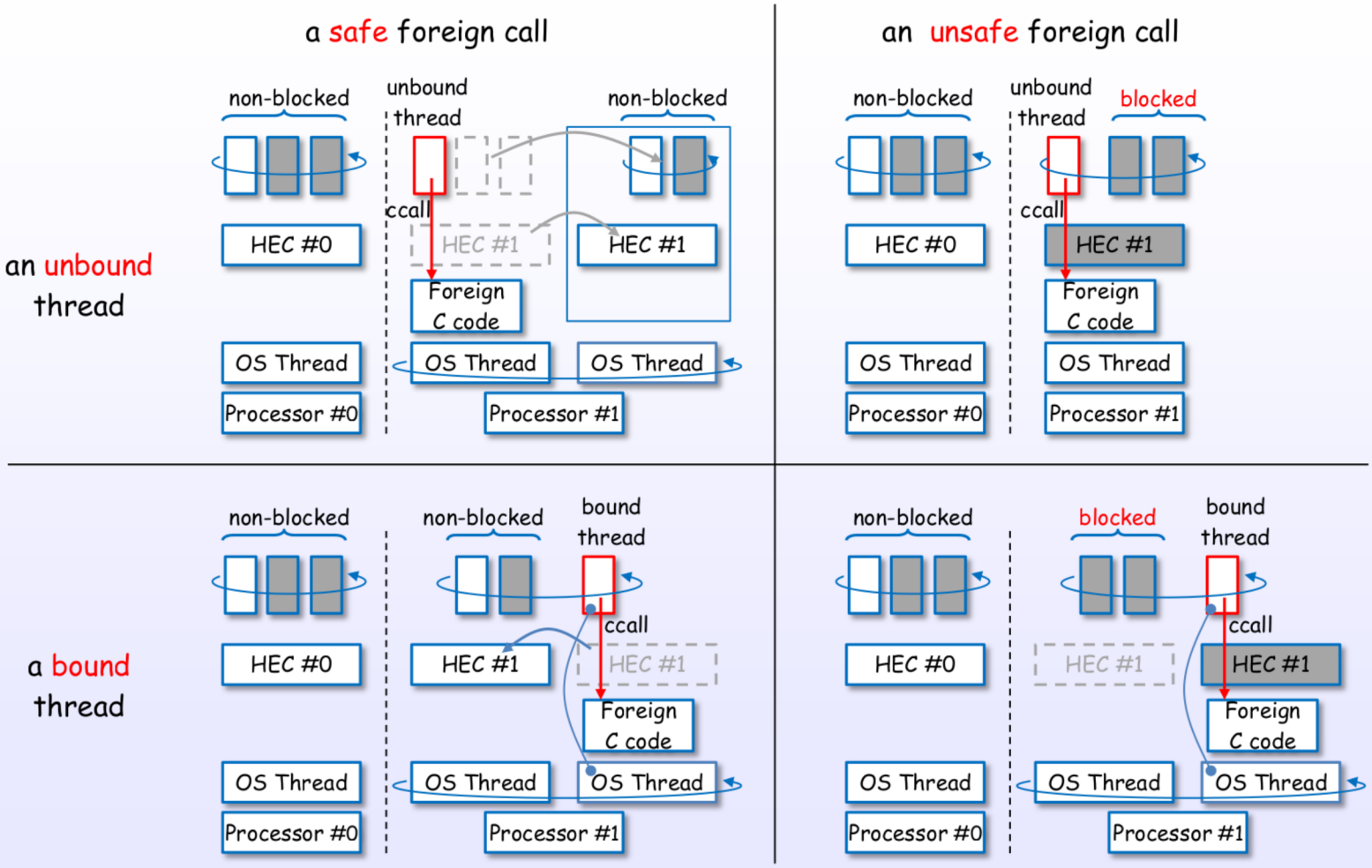
an unsafe foreign call



faster,
but blocking to the other Haskell threads

References : [6], [11], [20], [S39], [S38], [S37], [S36], [S40]

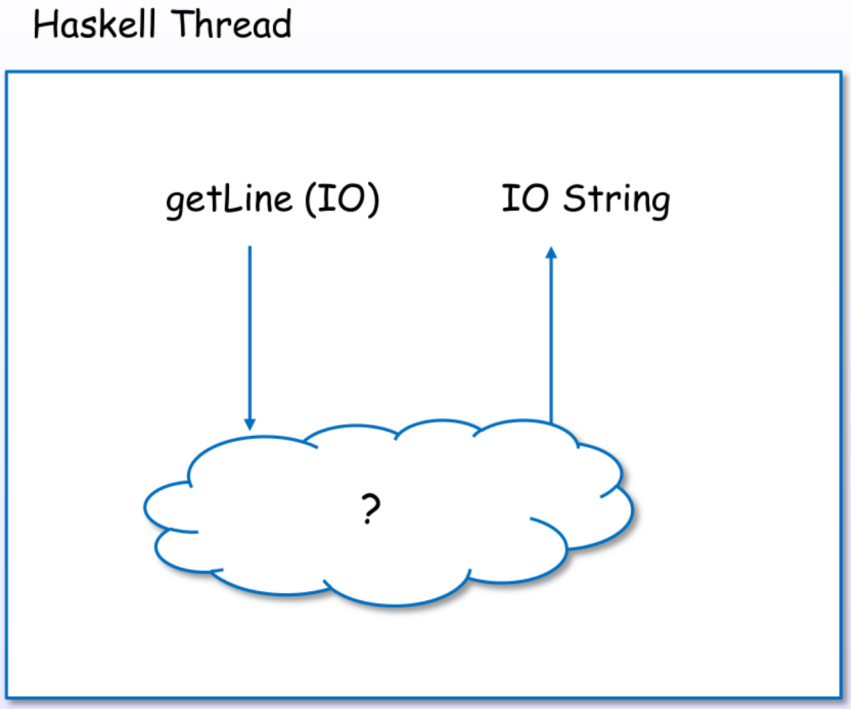
Safe/unsafe foreign call and bound/unbound thread



References : [6], [11], [20], [S39], [S38], [S37], [S36], [S40]

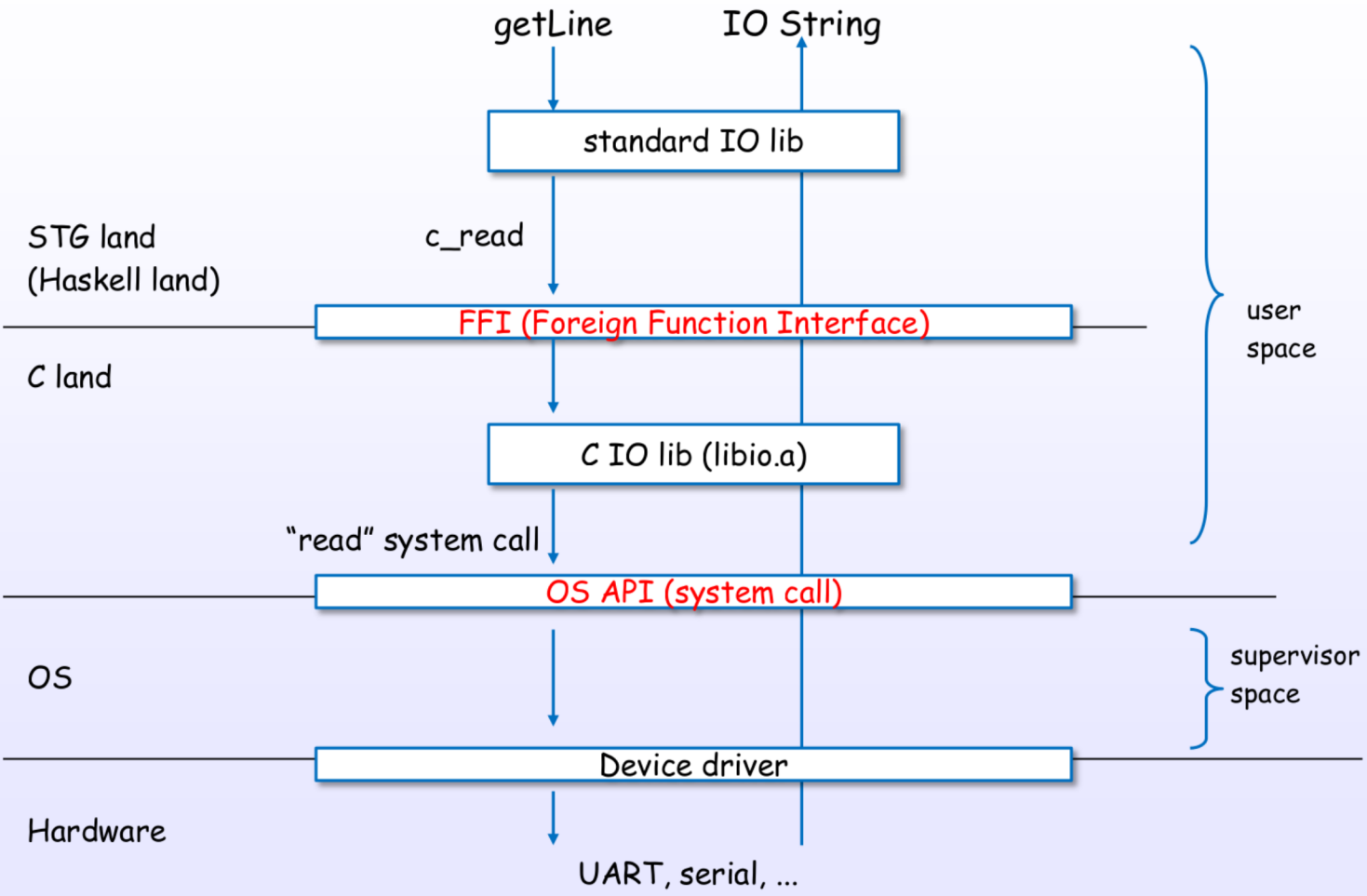
IO and FFI

IO



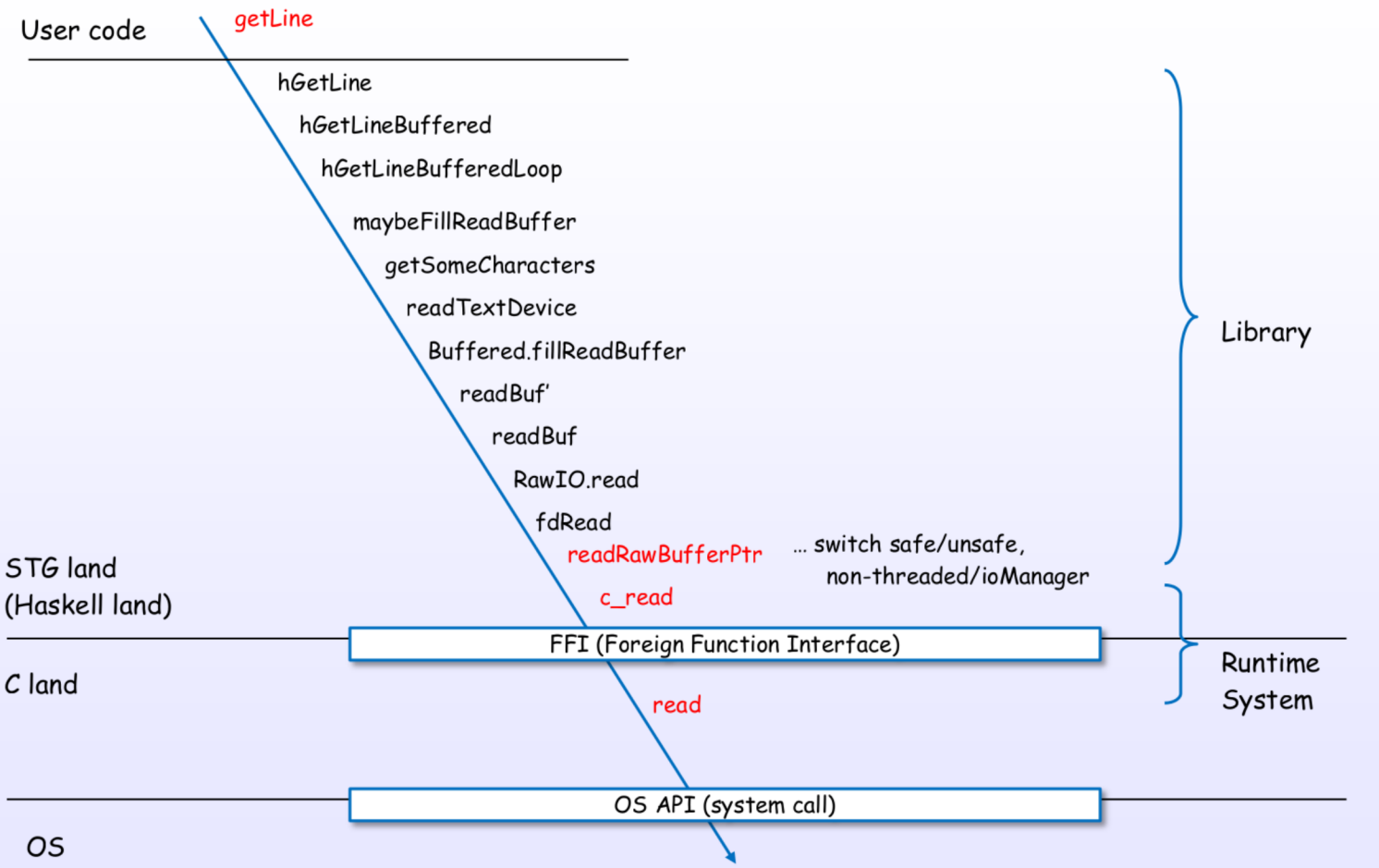
References : [6], [11]

IO example: getLine



References : [6], [11], [20], [S39], [S38], [S37], [S36], [S40]

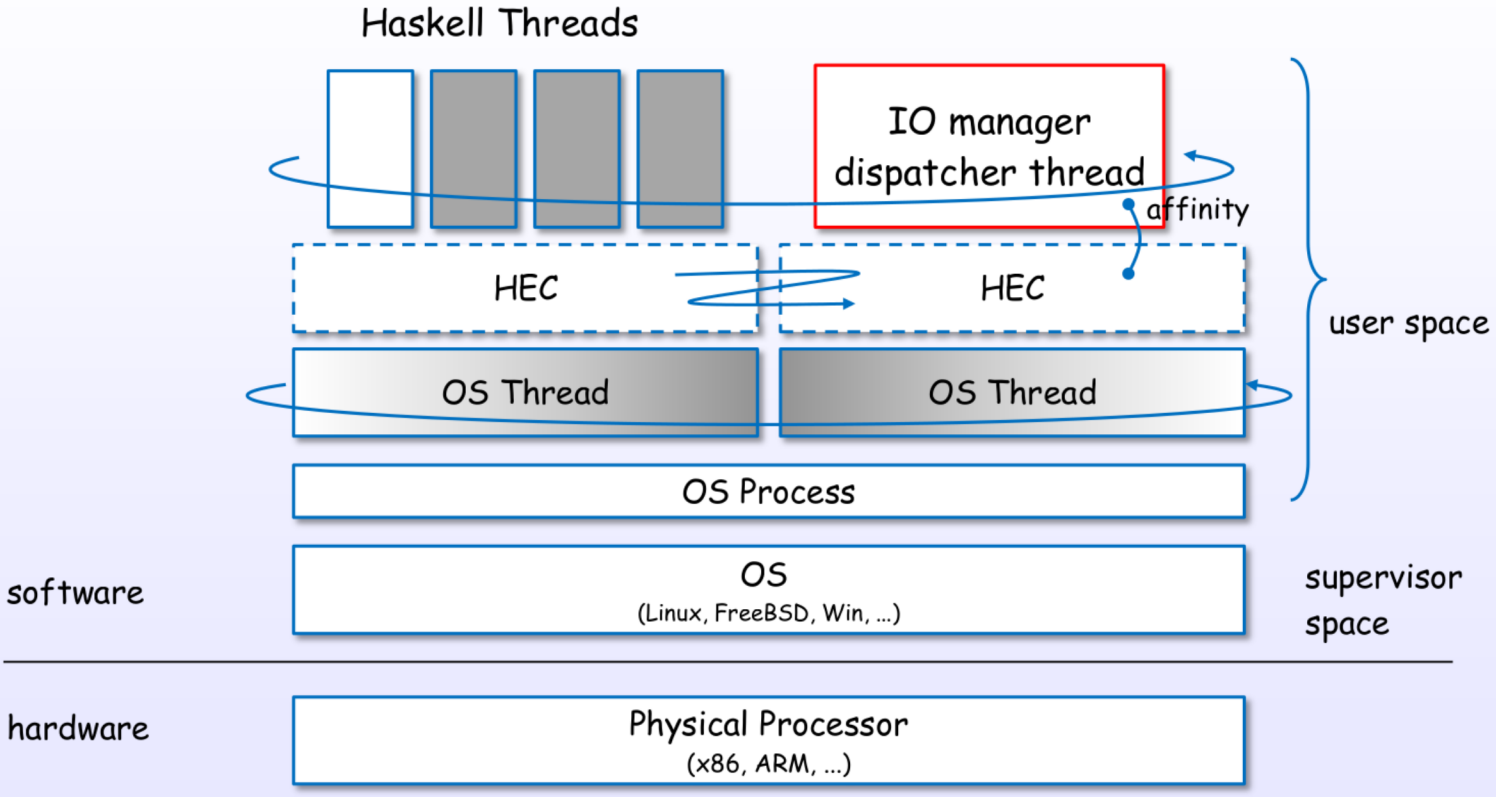
IO example: getLine (code)



References : [6], [11], [20], [S39], [S38], [S37], [S36], [S40]

IO manager

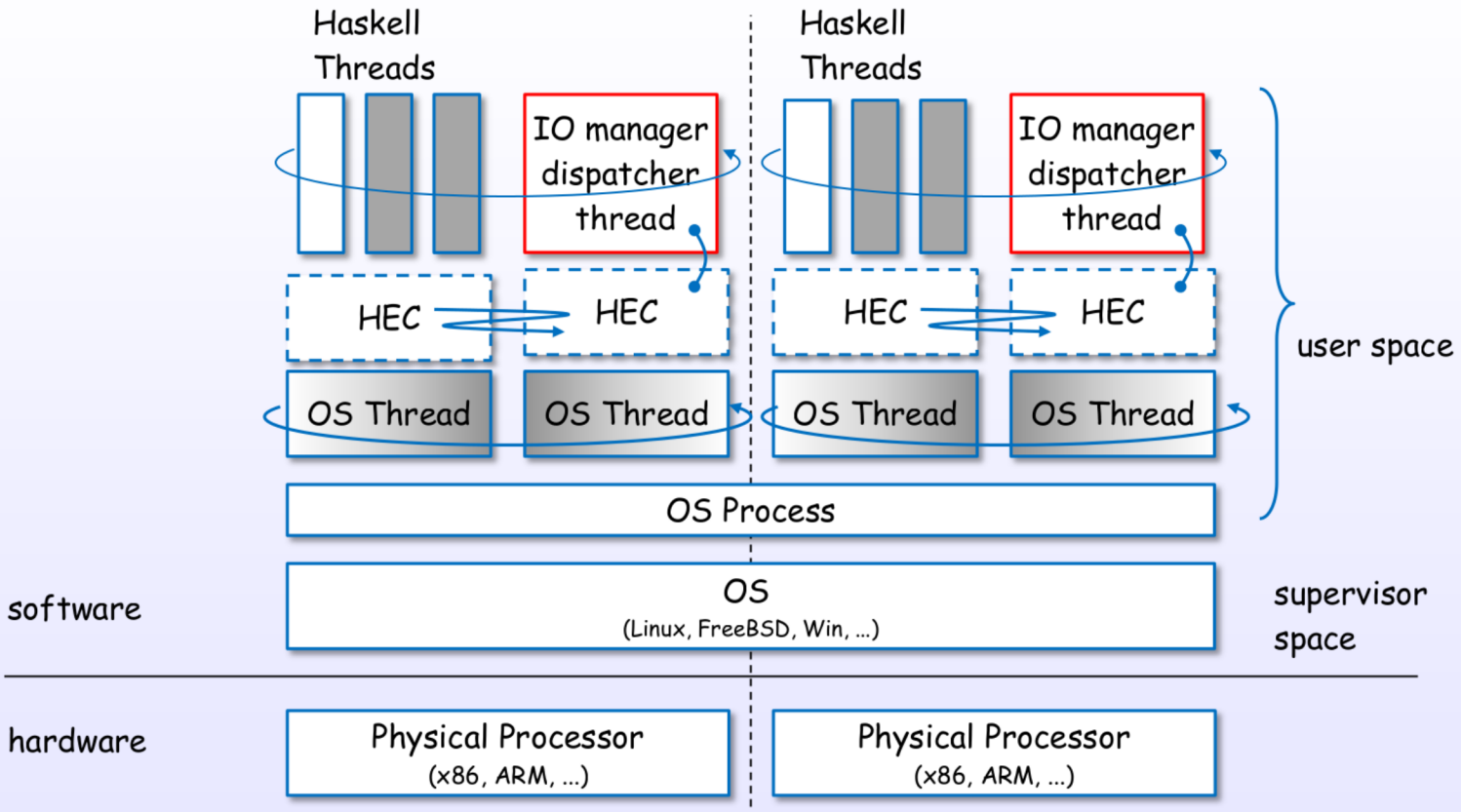
IO manager (single core)



*Threaded option case (ghc -threaded)

References : [7], [5], [8]

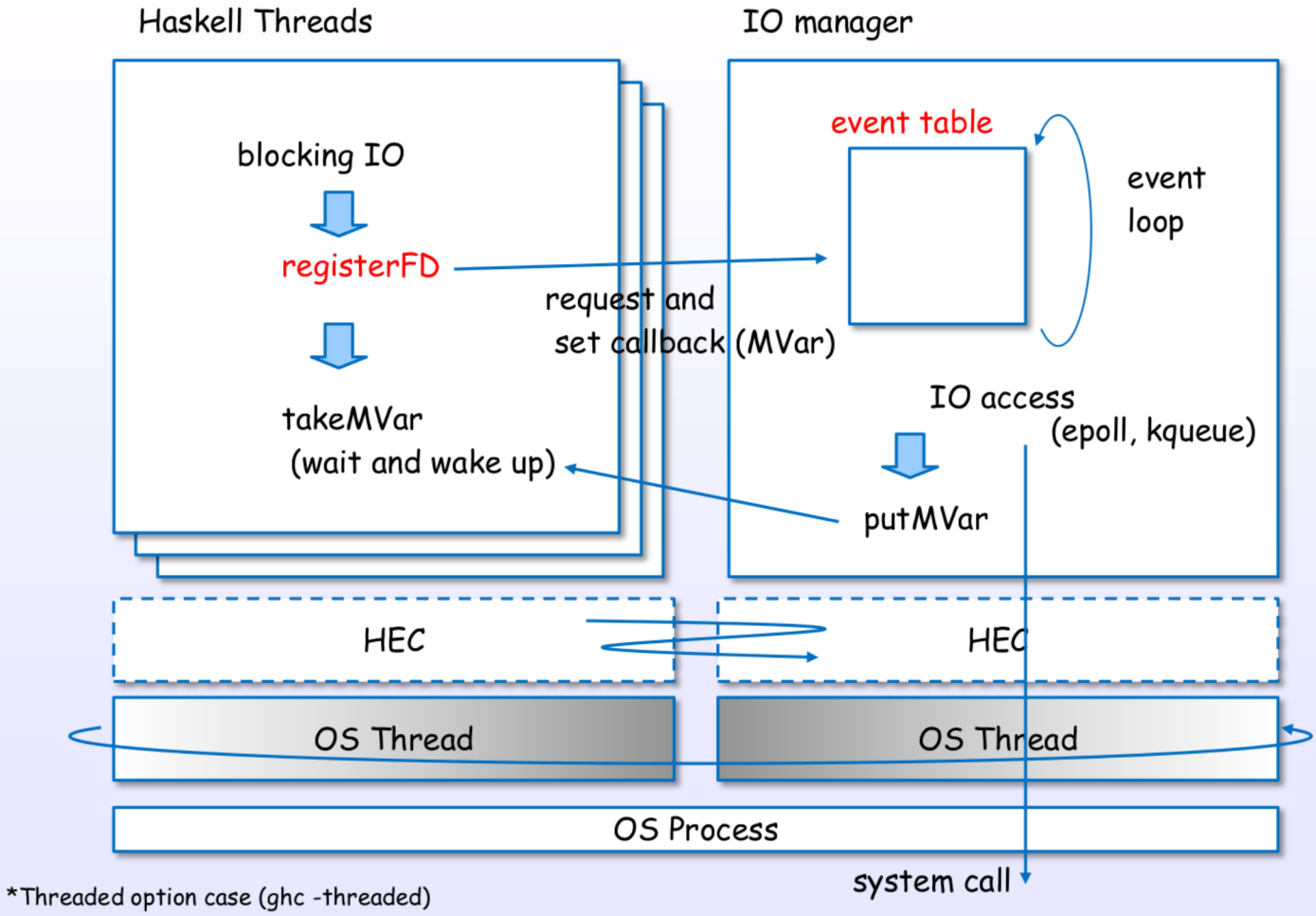
IO manager (multi core)



*Threaded option case (ghc -threaded)

References : [7], [5], [8]

IO manager

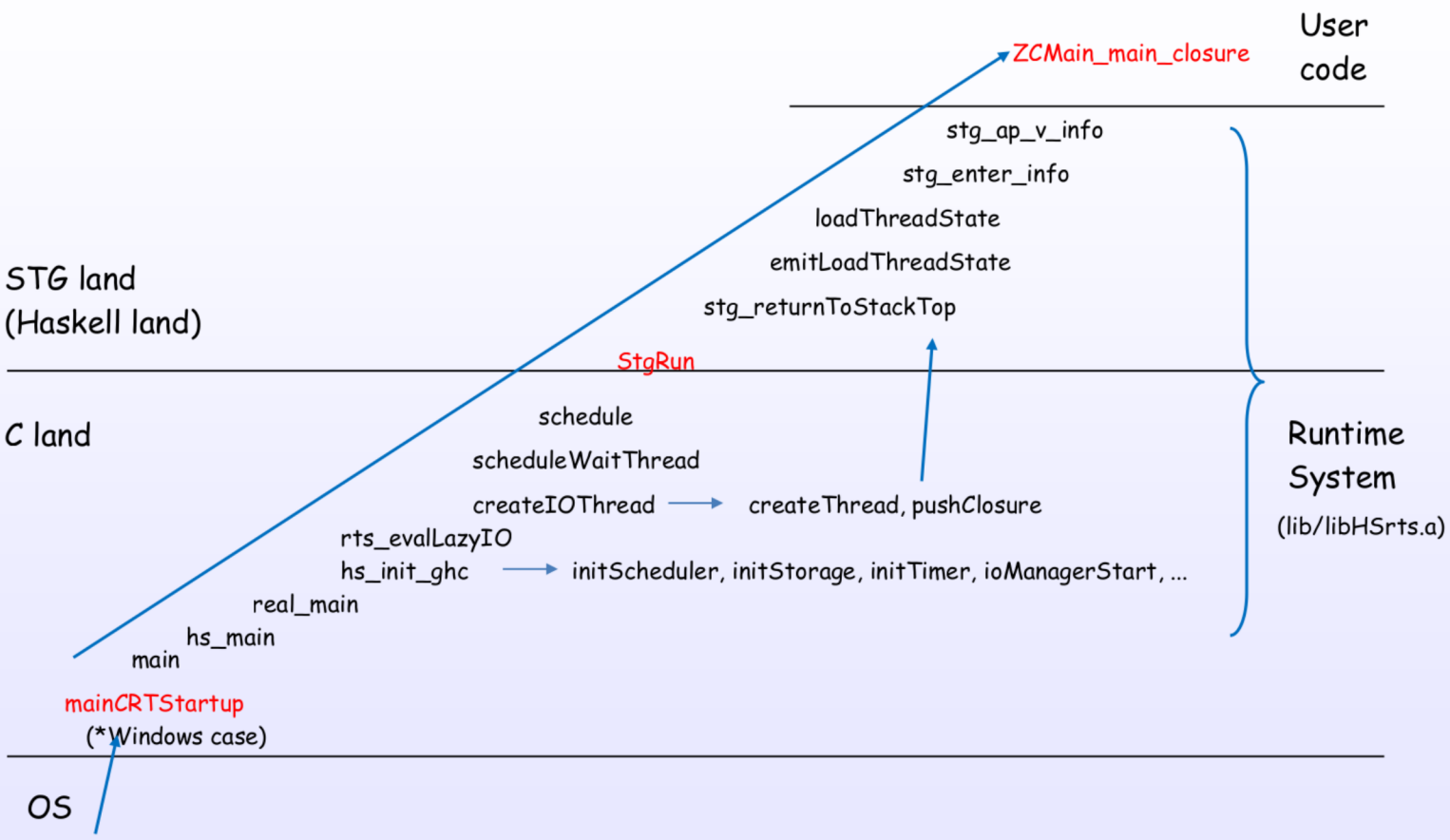


*Threaded option case (ghc -threaded)

References : [7], [5], [8], [S29], [S30], [S32], [S37], [S35], [S3]

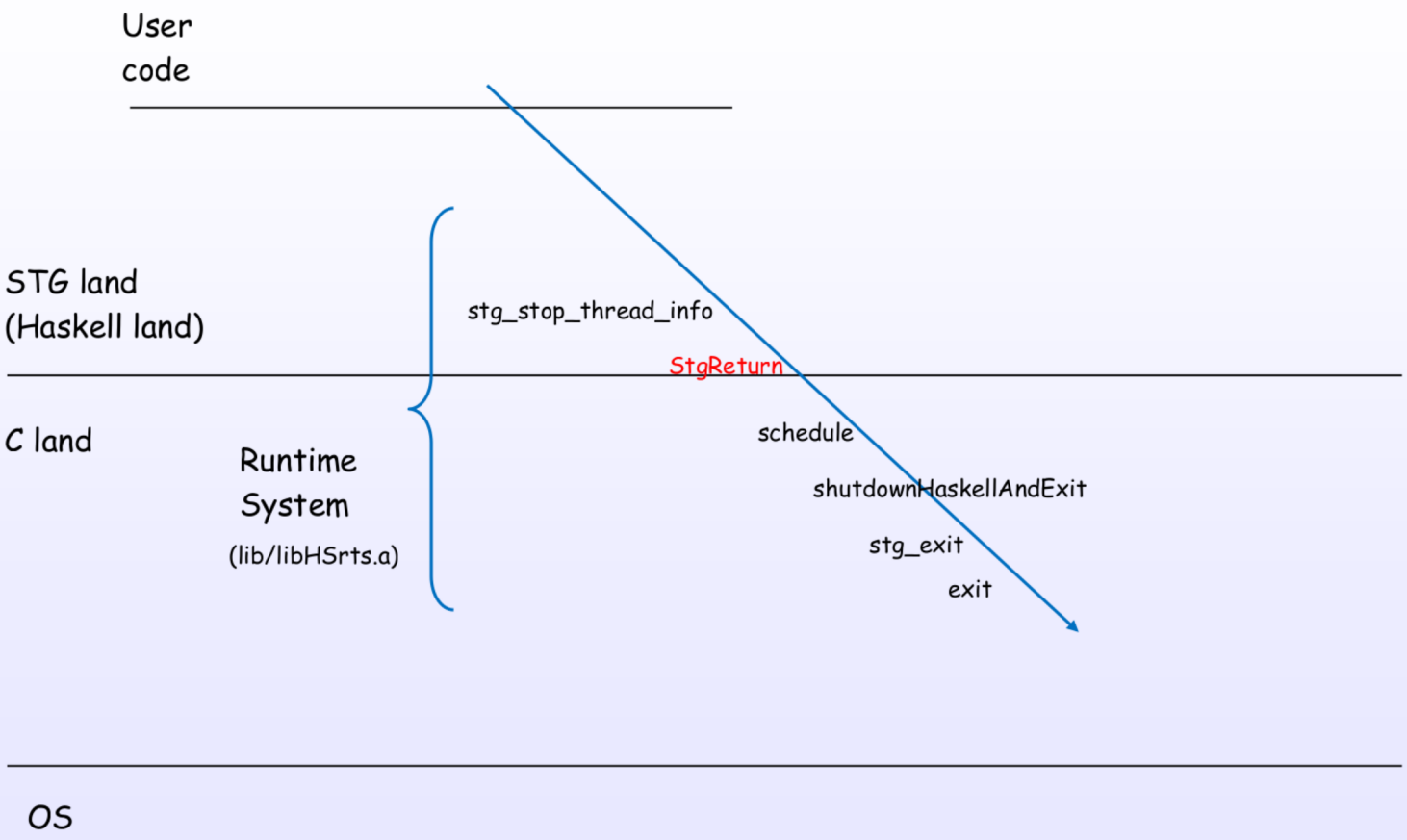
Bootstrap

Bootstrap sequence



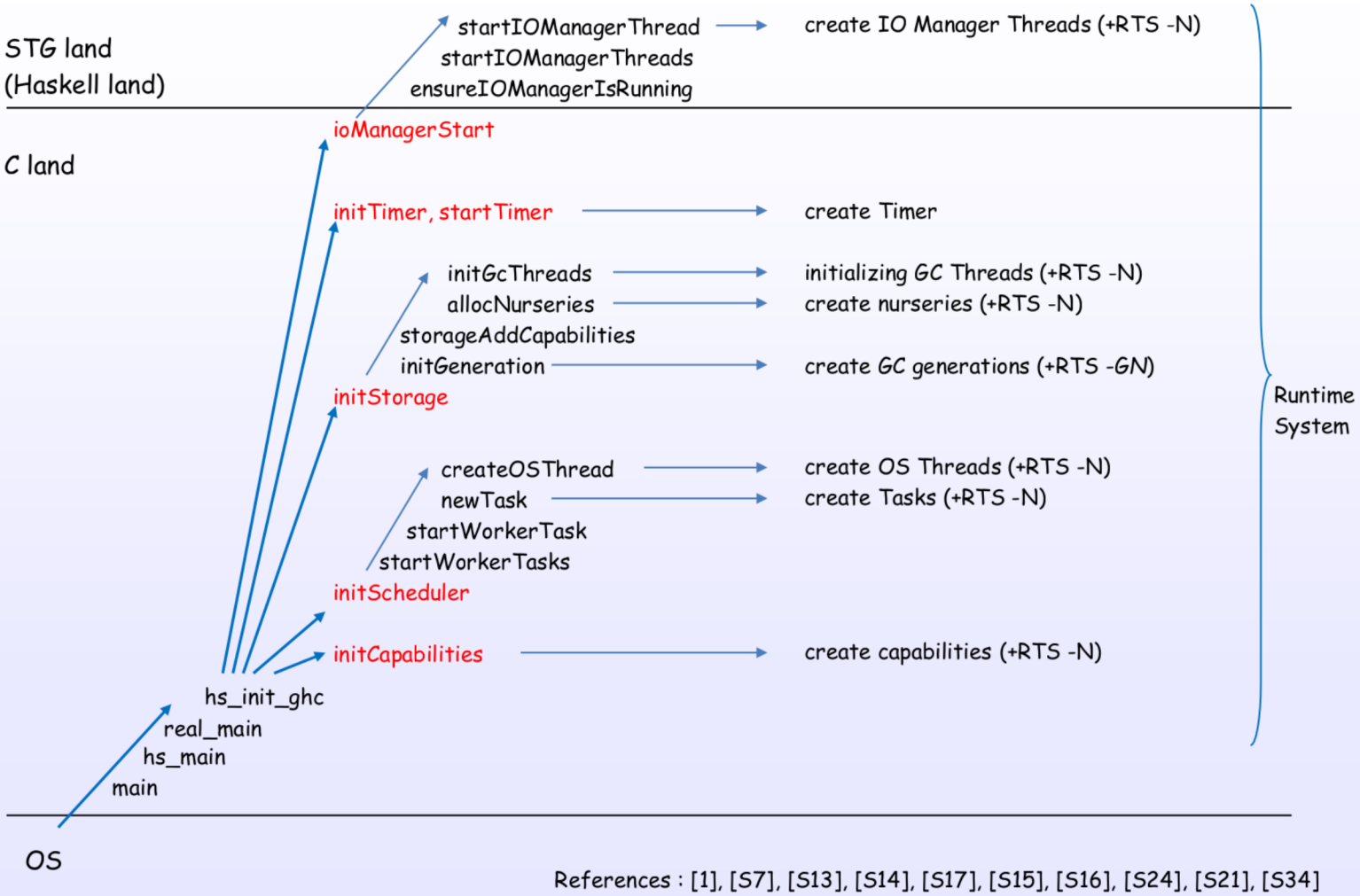
References : [S7], [S13], [S14], [S17], [S18], [S19], [S9], [S10], [S21], [S41]

Exit sequence



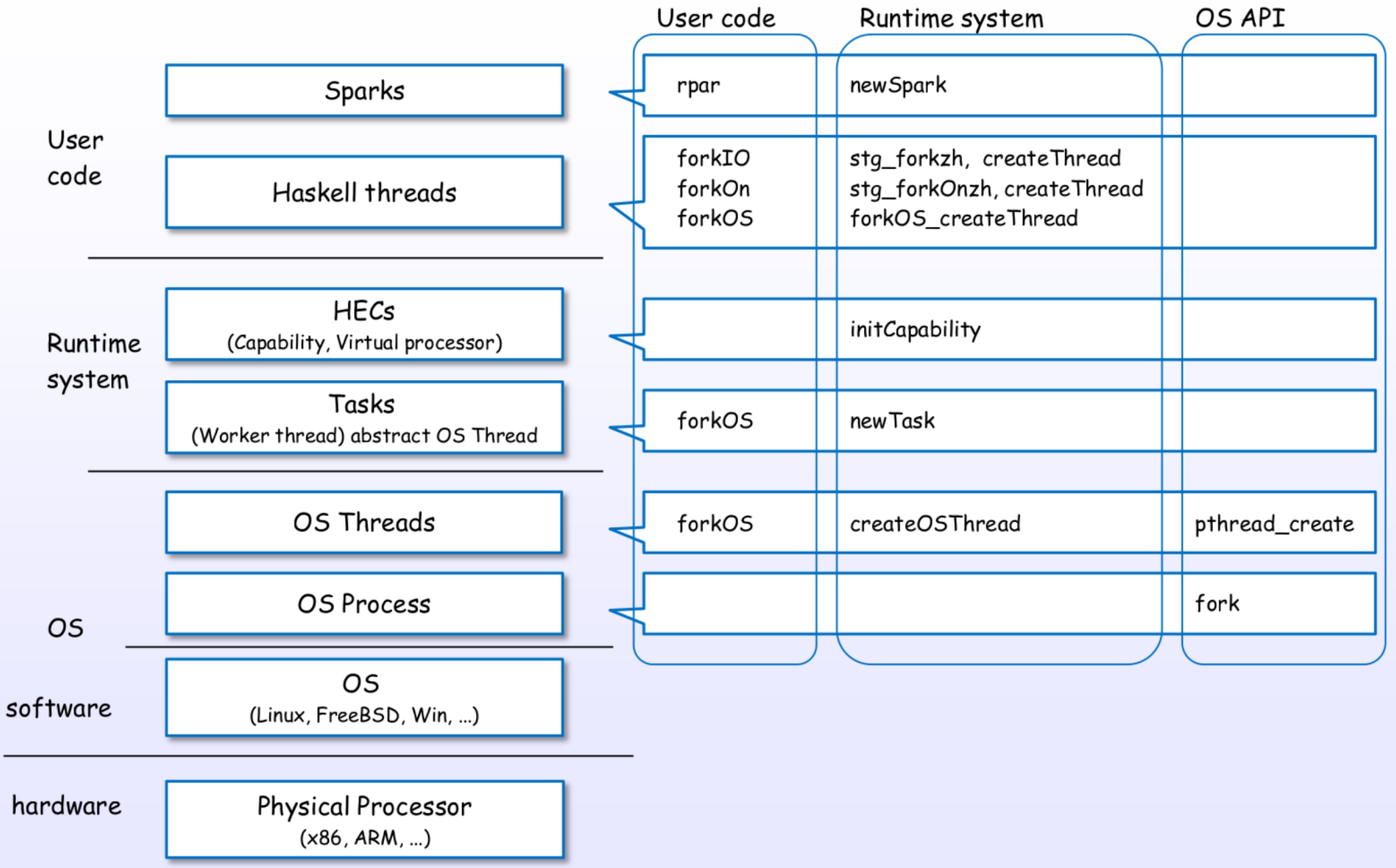
References : [S19], [S18], [S17]

Initializing



References : [1], [S7], [S13], [S14], [S17], [S15], [S16], [S24], [S21], [S34]

Create each layers



References : [1], [5], [8], [9], [C11], [C17], [S12], [S26], [S22], [S15], [S23]

References

References

[1] The Glorious Glasgow Haskell Compilation System User's Guide
https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/index.html

[2] Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine Version 2.5
http://research.microsoft.com/en-us/um/people/simonpj/Papers/spineless-tagless-gmachine.ps.gz

[3] Making a Fast Curry Push/Enter vs Eval/Apply for Higher-order Languages
http://research.microsoft.com/en-us/um/people/simonpj/papers/eval-apply/

[4] Faster Laziness Using Dynamic Pointer Tagging
http://research.microsoft.com/en-us/um/people/simonpj/papers/ptr-tag/ptr-tagging.pdf

[5] Runtime Support for Multicore Haskell
http://research.microsoft.com/en-us/um/people/simonpj/papers/parallel/multicore-ghc.pdf

[6] Extending the Haskell Foreign Function Interface with Concurrency
http://community.haskell.org/~simonmar/papers/conc-ffi.pdf

[7] Mio: A High-Performance Multicore IO Manager for GHC
http://haskell.cs.yale.edu/wp-content/uploads/2013/08/hask035-voellmy.pdf

[8] The GHC Runtime System
web.mit.edu/~ezyang/Public/jfp-ghc-rts.pdf

[9] The GHC Runtime System
http://www.scs.stanford.edu/14sp-cs240h/slides/ghc-rts.pdf

[10] Evaluation on the Haskell Heap
http://blog.ezyang.com/2011/04/evaluation-on-the-haskell-heap/

References

[11] IO evaluates the Haskell Heap
http://blog.ezyang.com/2011/04/io-evaluates-the-haskell-heap/

[12] Understanding the Stack
http://www.well-typed.com/blog/94/

[13] Understanding the RealWorld
http://www.well-typed.com/blog/95/

[14] The GHC scheduler
http://blog.ezyang.com/2013/01/the-ghc-scheduler/

[15] GHC's Garbage Collector
http://www.mm-net.org.uk/workshop190404/GHC's_Garbage_Collector.ppt

[16] Concurrent Haskell
http://www.haskell.org/ghc/docs/papers/concurrent-haskell.ps.gz

[17] Beautiful Concurrency
https://www.fpcomplete.com/school/advanced-haskell/beautiful-concurrency

[18] Anatomy of an MVar operation
http://blog.ezyang.com/2013/05/anatomy-of-an-mvar-operation/

[19] Parallel and Concurrent Programming in Haskell
http://community.haskell.org/~simonmar/pcph/

[20] Real World Haskell
http://book.realworldhaskell.org/